

第六章 XaaS 模式的第三方运营与优化

6.1 引言

在传统的分布式系统中，软件的交付方式通常是由用户一次性付费，获得软件的拥有权和使用权。用户购买软件后，还需要从头至尾建设支撑软件运行所需要的 IT 基础设施（关于 IT 基础设施的定义，请参见本书 1.3.2.3 节，它包括基础设施软件、硬件及其物理环境等），并自行进行软件的运维管理。对软件的购买者来说，这种软件交付和运营方式存在以下不足之处：用户在软件费用的支出之外，还需要在 IT 基础设施上进行一次性的大量投入；用户购买的 IT 基础设施资源无法得到充分利用；用户在系统运行维护方面需进行高昂的投入，且很难达到良好的效果；传统的软件交付方式缺乏灵活性，很难适应动态变化的业务需求，且导致用户在软件上的投资浪费和重复投资等。本书在基础篇 1.5.4 节对这些不足之处有详细的解释，在此不再赘述。

由前面讨论可见，传统的软件交付和运营方式虽然延续多年，但从长远角度来看，并不是最经济合理的方式。为此，人们开始探索一种新的软件交付和运营方式。“一切皆服务“(anything as a service, XaaS)模式的第三方运营正是在这种背景下兴起的。区别于传统的软件交付和运营方式，XaaS 模式把软件功能推到基础设施层面，由第三方运营商负责运维，并支持用户以“使用而不拥有”的方式消费和利用。在这种模式下，软件并非归软件的使用者所有，而是归专业的软件运营商（有时由软件的原始提供者担任运营商的角色）所拥有。软件运营商负责进行软件的部署和运维，根据用户的需要和实际使用情况来收费。这种交付和运营模式弥补了传统软件交付和运营方式的不足：(1) 用户不需要进行一次性大量投入。用户不需要购买软件的拥有权，也不需要进行 IT 基础设施的建设，而是以“Pay as you go”的方式支付一定的服务费用，这样，用户不需要进行一次性大量投入，在起步阶段可以通过小成本的投入规避项目风险。(2) 有助于提高 IT 基础设施资源的利用率。由于用户并不单独建设 IT 基础设施，而是由软件运营商将多个用户所需的计算和存储等资源集中起来进行管理，这就为软件运营商利用资源的调度和优化技术来提高资源的整体利用率提供了条件。(3) 用户无需在系统的运行维护方面投入，且可根据业务的需求进行 IT 投入的调整，在 IT 服务费用的投入方面享有更高的灵活性。

近年来，XaaS 模式的第三方运营正在得到越来越广泛的关注和认同，并不断得到厂商和客户的认可。XaaS 模式第三方运营和优化已经成为互联网计算的

一个重要组成部分。本章围绕 XaaS 模式的第三方运营和优化，给出基本定义之后，将分析以下两组核心问题：(1) XaaS 模式的第三方运营有哪几种交付方式？如何对它们进行分类？XaaS 模式下，软件的生命周期是什么样的，和传统软件相比有什么不同？如何度量 XaaS 模式的成熟度？(2) XaaS 的基本特征是什么？互联网服务的性质保障遇到哪些挑战性的问题？本章最后，将给出一种 XaaS 模式第三方运营的典型实例。

6.2 基本概念

本书将 XaaS 定义为以服务形式提供网络化的软件能力和资源的一种方式。从不同视角、不同内容和不同层面看问题，人们又赋予它多种多样的服务供给标识，典型的有：SaaS、PaaS、IaaS 及 DaaS 等。其中，离最终用户最近的是 SaaS。SaaS 作为一种应用软件（尤指可共享的 Web 应用）的部署、运营和使用模式，强调把应用软件统一部署在运营端（往往还会涉及到一到多个数据中心），用户则通过网络以按需付费等商业模式来使用应用软件，运营端和用户之间可达成细粒度的服务水平协议。在 PaaS 模式下，租户并非直接获取传统的应用软件服务，而是获取平台服务，例如在线开发环境等，租户在这些平台服务之上构造应用软件。在 IaaS 模式下，租户获取的是基础设施服务，例如虚拟机和存储服务等。事实上，在本书第三章所提出的 cSI 体系结构中，尽管服务的类型和层次千差万别，但无论是 SaaS、PaaS、IaaS 和 DaaS 等，它们都是基础设施对外以服务的模式提供资源的一种形式，因此，我们将其统称为“资源即服务”(resource as a service, RaaS)或“一切皆服务”(anything as a service, XaaS)。XaaS 模式下的软件对外常常以服务的形式提供，这些服务有网页、Web 数据库服务、RSS/Atom 种子、Open API 和 Web 服务等多种形式。

多租户是 XaaS 模式的核心概念之一。租户(tenant)指一个具有相同需求的最终用户群体，最终用户以租户为单位租用软件。多租户(multi-tenants)指能共享同一软件的多个租户群体。在 XaaS 模式下，运营端统一对应用软件需要的计算、存储和带宽资源进行多租户的资源共享和优化，并且能够根据实际负载进行性能扩展。负责 XaaS 模式应用软件运营的机构，通常称为 XaaS 运营中心。除了提供软件运维服务并确保用户可以通过互联网 24×7 小时不间断访问之外，XaaS 运营中心往往通过技术和管理等手段提升后台效率和共享能力，挤出盈利空间。

定义 6.1 租户(tenant): 在 XaaS 模式下，具有相同需求的最终用户群体被称作租户(tenant)。最终用户以租户为单位来租用软件。

定义 6.2 软件即服务(software as a service, SaaS): 一种应用软件（尤指可共

享的 Web 应用) 的部署、运营和使用模式。在 SaaS 模式下, 强调把应用软件统一部署在运营端 (往往还会涉及到一到多个数据中心), 用户则通过网络以按需付费等商业模式来使用应用软件, 运营端和用户之间可达成细粒度的服务水平协议。运营端统一对多个租户的应用软件需要的计算、存储和带宽资源进行资源共享和优化, 并且能够根据实际负载进行性能扩展。

定义 6.3 一切皆服务(anything as a service, XaaS): XaaS 是以服务形式提供网络化的软件能力和资源的一种方式, 它是软件即服务、平台即服务、基础设施即服务及数据即服务等统称。

定义 6.4 XaaS 运营中心: 负责 XaaS 模式应用软件运营的机构, 称为 XaaS 运营中心。

定义 6.5 XaaS 运营平台: 往往由一个或多个服务器集群构成, 是用以支撑 XaaS 模式应用软件运营的计算平台。

定义 6.6 第三方运营: XaaS 运营中心往往由不同于软件提供商和最终用户的第三方角色 (称为 XaaS 运营商) 来负责运行维护, 这被称为第三方运营。

6.2.1 XaaS模式下应用软件的运营模式分类

根据 XaaS 运营平台是否对第三方开放, 以及根据对 XaaS 模式下应用软件 (以下简称 XaaS 应用) 开发工作控制强度的不同可以对 XaaS 应用的运营模式进行分类。(徐志伟, 廖华明, 余海燕, 等., 2008)将网络计算系统对应用开发工作的控制进行了总结, 本章在此基础上进行了细化。运营平台对应用开发工作的控制主要通过以下几种形式体现: 命名、管理域、共享范围 (应用提供的服务和价值能为哪些用户使用)、接入、用户信息管理、计费以及用户访问权限管理等。

下面, 本章总结了四种 XaaS 应用的运营模式, 如表 6.1 和图 6.1 所示, 分别介绍如下:

1. 封闭型、完全集中控制模式 (“专卖店” 模式)

这种模式下 XaaS 应用的运营平台提供互联网应用开发的完全集中控制功能。控制权掌握在管理运营平台的某个组织手中, 应用的开发受该组织的统一规定、统一标准、统一管理和统一维护所约束。

这种模式下, XaaS 应用的运行完全托管到平台中, 托管的方式可以是开放托管方式和专有托管方式。开放托管方式是指运营平台可以运行在互联网上, 用户可以通过企业外部网络进行访问。专有托管方式是指运营平台运行在企业内部的专有网络中, 外部网络不能访问, 这通常是出于保障安全性的考虑。此外, 这种 XaaS 应用支撑平台并不对外开放增值应用开发接口, 无法使用支撑平台的功能和数据来开发第三方增值应用。

删除的内容: 表

删除的内容: 图

表 6.1 XaaS 应用运营模式总结

应用的托管方式	支撑平台的功能	应用场景	案例	
封闭型、完全集中控制模式(“专卖店”模式)	平台并不对外开放增值应用开发接口, XaaS 应用的构造和运行由平台完全控制	对平台的多租户、可配置、可伸缩性以及高可用性、高可靠性要求高	集团企业、中小企业或行业应用的信息化	Salesforce
开放型、接入及管理集中控制模式(“中介店”模式)	平台并不支撑 XaaS 应用的实际运行, 而是通过平台开放接口控制应用的一些基本功能	平台须支持对已有软件的接入、计费、命名和分类管理等功能	电子商务平台	阿里软件
开放型、完全集中控制模式(“超市”模式)	平台开放增值应用开发接口并对 XaaS 应用具有完全的控制能力	平台提供基础服务及业务服务, 提供应用开发接口, 对平台可伸缩性、高可用性和高可靠性要求高	集团企业、中小企业或行业应用的信息化	Force, Zoho Marketplace
开放型、分散控制模式(“Mall”模式)	并不能完全控制对应用的管理, 而是交给第三方更大的控制权	平台提供更基础的服务和应用开发环境, 对平台通用性要求高, 对存储和计算资源虚拟化能力要求高	各种企业级 Web 应用和电子商务等	Windows Azure Amazon Web Services Google AppEngine

这种模式的优点是：用户无需运维 XaaS 应用，由于 XaaS 应用由平台完全控制，为运营商进行资源充分的优化配置和管理提供了条件。这种模式可应用于集团企业、中小企业或行业应用的信息化场景中，其代表性工作包括 Salesforce.Com(<http://www.salesforce.com>)。

2. 开放型、接入及管理集中控制模式（“中介店”模式）

这种模式下，XaaS 运营平台并不支撑 XaaS 应用的实际运行，但是，对应用的接入、用户信息管理和计费等一些基本的功能进行集中控制。这些功能是通过 XaaS 运营平台的开放接口对外提供的，可以使用这些接口开发相应的功能，来构造新的 XaaS 应用，将其接入到平台中，或者对 XaaS 应用的使用进行计费。同时，开发者在命名、管理域及共享范围等其他一些方面则享有控制权。

这种模式可以应用于电子商务平台等场景中，其代表性工作如阿里软件(<http://www.alisoft.com/>)等。

3. 开放型、完全集中控制模式（“超市”模式）

这种模式下，XaaS 运营平台是对外开放的，通过对外提供应用开发接口来提供一些基础服务及业务服务，从而软件开发者和增值服务开发者可以在一定范围内开发插件、进行应用定制和开发集成应用等。

这种模式下，平台对 XaaS 应用的开发具有完全的控制能力，控制权掌握在管理运营平台的某个组织手中，应用的开发受该组织的统一规定、统一标准、统一管理和统一维护所约束。

由于平台的开放性，这种模式必须应对不可预测的用户负载，其对平台的多租户、可配置、可伸缩性、可用性以及可靠性要求也较高。这种模式可应用于集团企业、中小企业或行业应用的信息化典型场景中，其代表性工作包括 Salesforce 公司开发的 Force.com 平台和 AdventNet 公司开发的 ZOHO 网络办公套件(<http://www.zoho.com>)。

4. 开放型、分散控制模式（“Mall”模式）

这种模式下，XaaS 运营平台是对外开放的，软件开发者和增值服务开发者可以利用平台所提供的基础服务开发新的互联网应用。

不同于“完全控制”模式的 XaaS 运营平台，这种模式下平台虽然控制着应用的运行，但其对应用开发的控制权则分散在各个开发者手里，每个开发者可自行获取所需的信息，自主决定命名、管理域、共享范围、计费、用户信息管理和访问权限控制等重要内容。

由于平台的开放性，这种模式下支撑平台也要提供基础服务、应用开发环境

以及很多保障性的功能。这种模式对平台通用性要求高，对存储和计算资源虚拟化能力要求高。这种模式可应用于各种企业级 Web 应用和电子商务等场景中，其代表性工作包括 Amazon Web Service(<http://aws.amazon.com>)、Microsoft Windows Azure(<http://www.microsoft.com/windowsazure/windowsazure/>) 以及 Google AppEngine(<http://appengine.google.com>)等。



图 6.1 按照开放和控制两个维度划分的 XaaS 应用运营模式

6.2.2 XaaS模式概念模型和软件生命周期

本节首先以提供平台服务的 PaaS 模式(XaaS 的平台服务提供模式)为例，介绍 XaaS 模式下软件的开发、部署及管理各个阶段所涉及的角色。如图 6.1 所示，将 XaaS 模式涉及到的角色抽象为一个“三角架构”模型，该模型由服务提供者、服务消费者和服务运营者三类角色组成。在图中，实线箭头表示角色的行为，虚线箭头表示角色的转换。其中，在实际的 XaaS 系统中，独立软件开发商 (independent software vendor, ISV) 是常见的服务提供者；服务消费者是增值服务开发商或最终用户；第三方服务运营商是常见的服务运营者。服务提供者提供基础服务（如消息通信、目录、安全和记账等共性的支撑类服务）、业务服务（对应行业业务相关的功能抽象或具体实现）和增值服务（如基于基础服务和业务服务的各类组合服务）。

删除的内容: 图

早期的做法是由服务提供者负责将服务发布到运营中心，并托管至运营中心由服务运营者进行服务的运行维护。后来，也发展到消费者也可以贡献服务，而运营中心也可能主动收集服务。服务运营者负责对这些服务进行第三方的运营和优化。

服务消费者从运营中心查找自己所需的服务，并和服务运营者签订服务水平

协议，然后，就可根据服务水平协议来使用服务。

在这个三角架构中，服务消费者也可以通过服务的组合等手段来构造新的服务，这时，服务消费者同时又具有服务提供者的角色。增值服务开发商就是具有服务消费者和服务提供者双重角色，它一方面通过集成或配置各种基础服务、业务服务，提供新的增值服务，另一方面基于 XaaS 平台所提供的各种数据、应用访问和配置 API，开发新的平台服务，扩展或增强原有 XaaS 平台的功能。

相比于本书基础篇第二章讨论的传统 SOA 概念模型（又称为 SOA 三角架构），XaaS 的“三角架构”有如下几点显著的区别：

(1) 服务消费者通过开发增值服务可转换为服务提供者；

(2) 在传统的 SOA 三角架构中，服务提供者仅仅将服务的元信息注册到注册中心，服务的运行和维护仍旧由服务提供者来负责，但是，在 XaaS 模式的三角架构中，服务提供者不仅将服务的元信息注册到运营中心，还将服务托管至运营中心由服务运营者进行运行维护，服务的实体实际运行在运营中心；

(3) 服务消费者通过和运营中心的交互来使用服务，不再需要和服务提供者进行交互。相对于 SOA 三角架构中的注册中心，服务运营者和运营中心在整个运作模型的重要性变的更加突出。

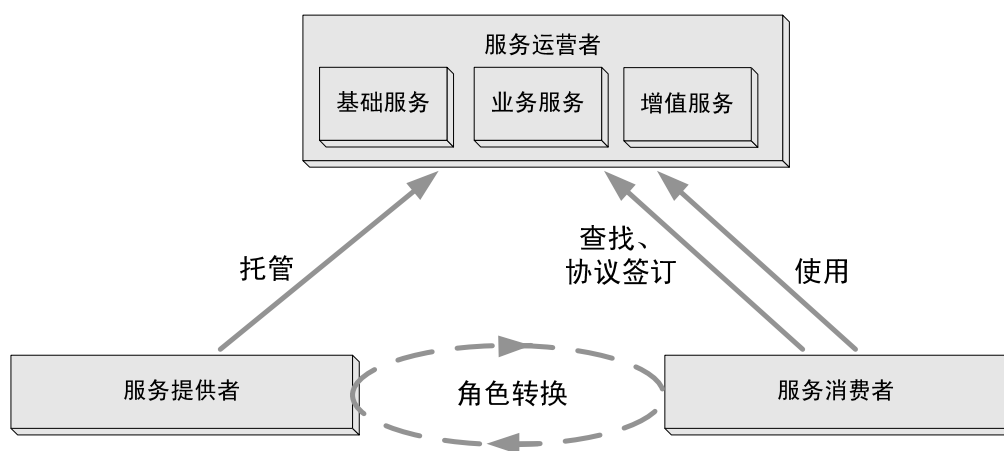


图 6.2 XaaS 模式概念模型

在 XaaS 模式概念模型中，各角色所需要解决的具体问题各有不同。服务运营者所负责的 XaaS 运营平台首先需要提供分布式计算环境的一些共性基础设施服务，例如通信、命名、同步和容错等服务。其次，需要将托管应用的一些共性服务抽象出来，在 XaaS 运营平台中实现，并尽可能提升抽象的层次和粒度，提供例如应用配置、运行时异常处理、日志和计费等共性服务。运营平台还提供细粒度的服务管理和仓储，例如服务的版本、目录管理和部署等。XaaS 运营平台的服务还需要通过 API 形式提供出来，支持通过浏览器等各种客户端进行在线

或离线的使用。服务提供者面临的主要挑战则包括如何基于现有的服务，开发新的应用或服务；以及如何基于现有 XaaS 运营平台所提供的 API 扩展平台的功能，开发新的服务，如何提供有服务质量（例如服务的 QoS 管理和服务推荐等）保障的服务，如何提高服务的互操作和易用性。

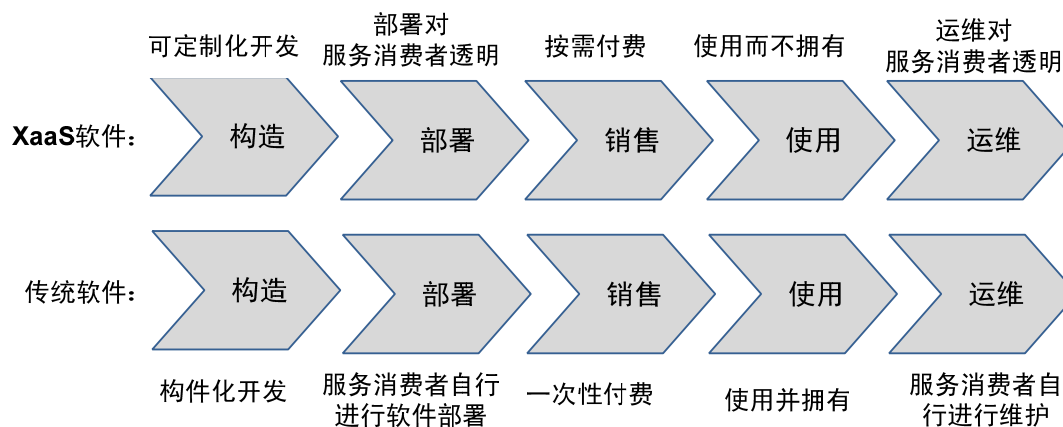


图 6.3 XaaS 模式软件生命周期和传统软件生命周期的对比

下面我们以基于平台服务创建的软件为例，来分析 XaaS 软件的生命周期。在软件的构造阶段，XaaS 平台提供应用开发的 API 和工具等，服务提供者利用平台所提供的这些 API 和工具等来开发应用。软件的部署也由服务提供者或服务运营者来操作，运营中心负责验证软件的可靠性和合法性，并落实软件的安装。在软件的销售阶段，租户发现运营平台上所有待售的应用软件，并可进行在线或离线试用，并根据按需付费协议进行购买。在软件使用阶段，租户可以对软件进行定制，运营中心可以为租户提供全套的定制解决方案。在使用过程中，XaaS 运营平台负责应用运行情况的监控和管理。在软件的维护阶段，服务提供者委托运营中心进行软件的升级维护，运营中心则负责落实软件的升级、重新部署和维护。

根据上面的分析可见，XaaS 模式下软件的生命周期和传统软件的生命周期还是有着明显区别的。

(1) 在软件构造阶段，XaaS 运营平台提供应用开发的 API 和工具等，这些 API 和工具一般都能够支持软件开发人员对应用进行定制化开发。而传统软件开发人员最多利用一些第三方提供的软件开发工具和开发包进行开发，这些第三方软件开发工具和开发包所提供的定制化能力是极为有限的。

(2) 在软件部署阶段，XaaS 模式下，软件的部署由服务提供者来操作，服务消费者无需自行进行软件的部署操作，而传统软件的用户必须自行进行软件的部署。

(3) 在软件销售阶段，传统软件采用的是一次性购买的收费模式，日后的软件维护另外收费，无形中增加了软件的成本。而 XaaS 模式下，采取的是根据用

户使用时间及占用资源的多少等具体情况进行按需付费的模式，不会另外加收软件维护费用。

(4) 在软件使用阶段，用户使用软件但不拥有软件，软件的功能和数据实际放在服务运营者那里，这和传统软件使用并拥有软件的方式完全不同。

(5) 在软件的运维阶段，XaaS 模式下的软件维护是对租户透明的，而传统软件的用户将必须自己或委托软件开发人员对软件进行维护。

6.2.3 XaaS模式下应用软件的体系结构

XaaS 模式下的应用软件以一些多租户和运营支撑相关的服务为基础，如多租户元数据服务、多租户安全服务、运营支撑服务和业务支撑服务等。图 6.4 示出了一个最基本的 XaaS 应用软件体系结构，一般来说，XaaS 应用软件可分为两部分：业务应用子系统和运营子系统。

删除的内容: 图

其中，业务应用子系统自下而上分为数据层、业务逻辑层和呈现层（和传统的分布式系统一样，对于分布式的 XaaS 应用，还应该将中间件层考虑在内）。数据层负责 XaaS 应用数据的存储和管理，一般包括元数据库、用户数据库和业务数据库等；业务逻辑层负责实现各租户的业务逻辑和业务流程；呈现层负责生成各租户的交互界面。

XaaS 应用软件的运营子系统包含的服务可分为两类：

(1) 运营支撑服务(operational support services, OSS)，处理软件运营方面的问题，包括帐户激活、提供、服务确保、服务消耗和计量等方面的问题。运营支撑服务主要用于精确地跟踪客户的资源使用状况，根据使用时间或资源多少进行精确的计费；根据租户定制的协议，在一定时间或对某些资源进行访问限制；对站点的访问状况和性能进行监控，保证其用户的 SLAs。

(2) 业务支撑服务(business support services, BSS)，主要支持计费（包括计价、评级、估税和款项代收等）以及客户管理（包括订单输入、客户自助服务、客户维护、故障登记和客户关系管理等）。

XaaS 应用对每个租户都必须预先提供一个具有管理权限的用户，并分配一个惟一的租户 ID。该管理用户可以创建其托管应用的最终用户帐号。这些功能是由多租户安全服务提供的。最终用户信息都带有其所属租户的租户 ID 标记，并通过目录服务存放于用户数据库中。目录服务和多租户安全服务还负责最终用户的身份认证和授权服务。

租户 ID 信息通过多租户元数据服务存放于元数据库中。XaaS 应用通常被设计为参数化的，在基本功能集之上，具有丰富的配置选项。每个租户的管理用户可通过配置应用选项来定制应用的呈现、用户接口、业务规则、工作流和安全策

略等。这些属于特定租户的元数据通过多租户元数据服务也存放于元数据库中。业务逻辑层和呈现层的模块通过多租户安全服务和多租户元数据服务得到租户的 ID 及其元数据信息，根据这些信息来执行相应的功能，其执行结果也根据租户 ID 隔离开来。

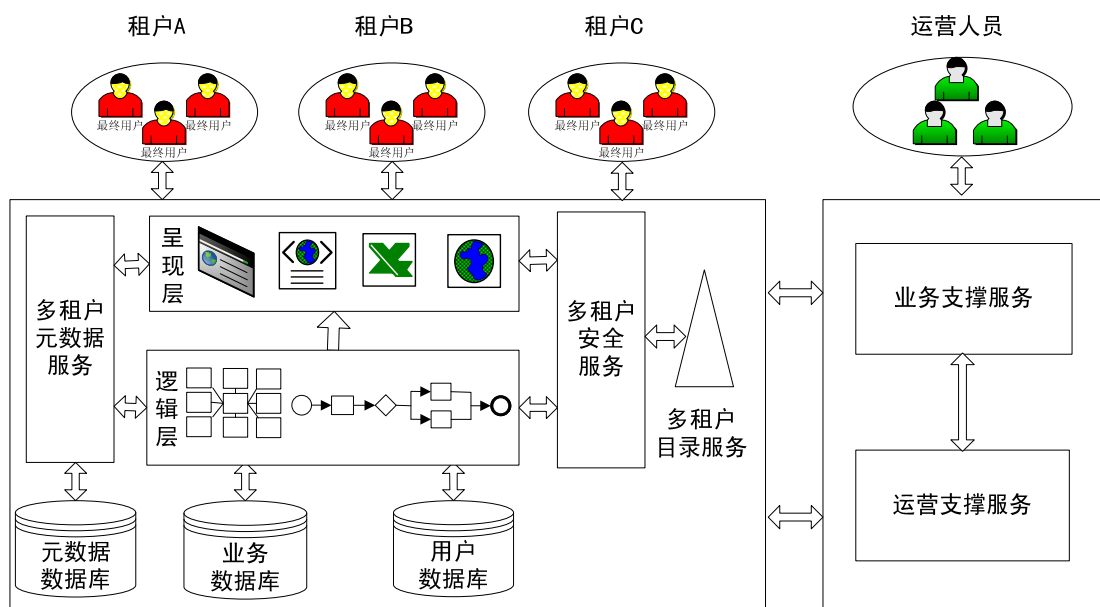
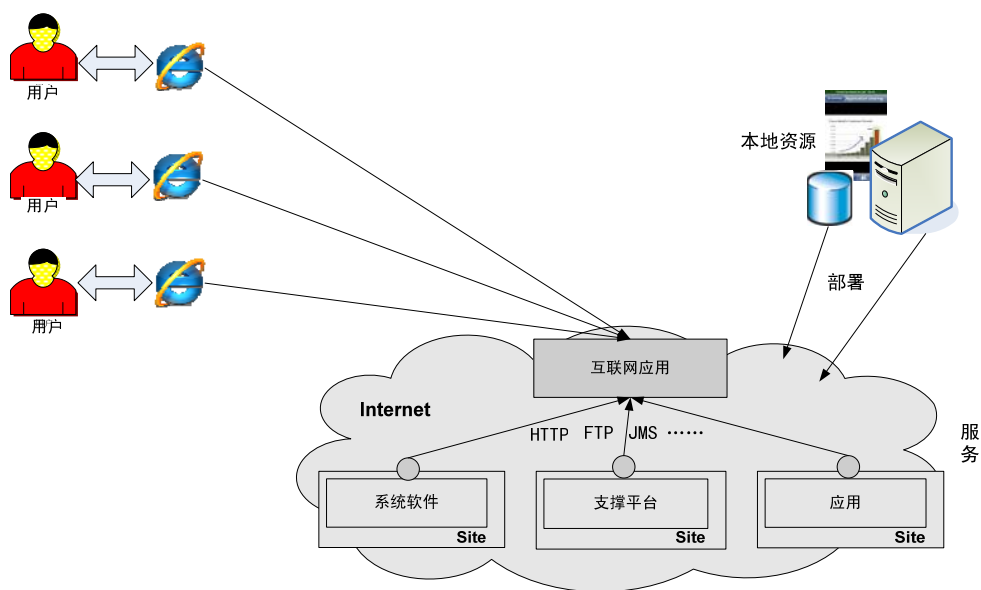


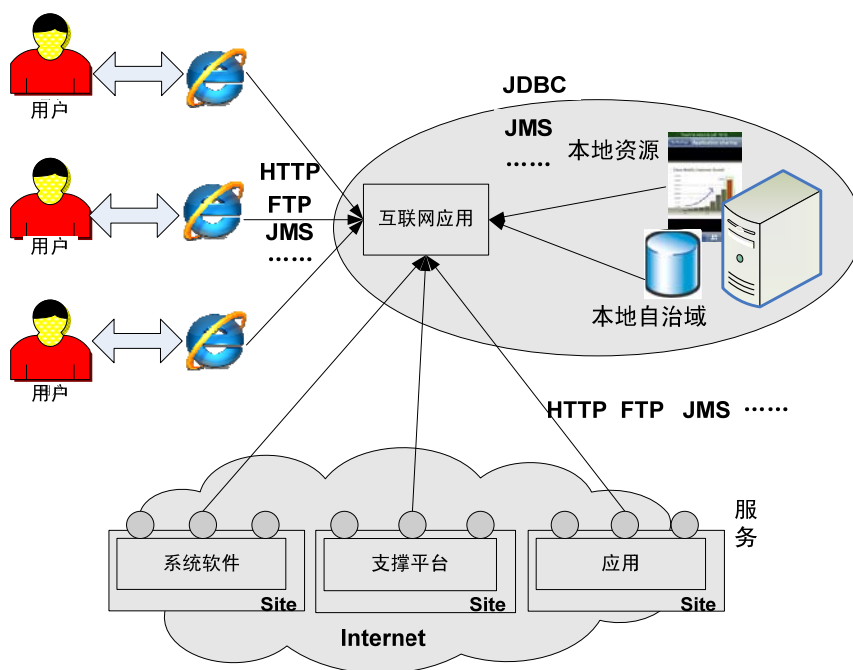
图 6.4 XaaS 应用的典型体系结构

从“客户端-服务器”体系结构的视角来看，XaaS 模式的应用软件一般来说可分为两部分，一部分为服务端程序，属于向租户提供服务的 XaaS 软件提供商的管理域，一部分为本地资源和应用程序，它们属于租户的管理域。若本地资源、应用程序以及数据部署到服务端并进行统一的管理和控制，便称为服务端计算，如图 6.5(a)所示。若应用软件的管理和控制在于租户的本地资源和程序所在的本地自治域内，则称为本地计算，如图 6.5(b)和图 6.5(c)所示。在本地计算模式下，若应用软件的管理和控制在于浏览器实现，则称为客户端计算，如图 6.5(c)。客户端计算有利于提高应用程序的效率和灵活性。

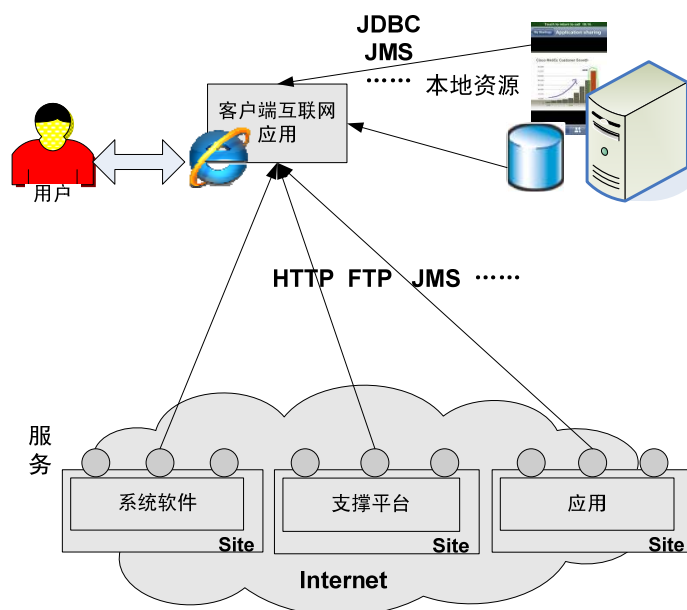
删除的内容: 图
删除的内容: 图
删除的内容: 图
删除的内容: 图



(a) 主控在服务端



(b) 主控在本地



(c) 主控在客户端

图 6.5 XaaS 应用的几种体系结构

6.3 XaaS模式第三方运营与优化的基本特征

前面已经讨论过,在 XaaS 模式下,最终用户以租户为单位来租用软件。XaaS 软件与传统软件最显著的不同就在于它支持多租户(multi-tenant-efficient)。XaaS 模式的多租户要求首先要保证租户之间在逻辑上的隔离。所谓隔离,是指在理想状态下,每个租户在使用 XaaS 软件时都彼此不受干扰,认为自己独享 XaaS 软件。例如,租户期望能够实现如下几点:针对各自的业务需求及其变化,可以在互不影响的前提下修改软件的数据模型;尽管其他租户对软件的利用率在随时变化,但是自己获得的服务质量始终有所保障;尽管与其他租户共享了同一软硬件平台,但是系统的安全能够得到保证。对分布式的 XaaS 应用来说,可以在应用层、中间件层、系统软件层和硬件层等不同的层次上实现多租户的隔离,下面将在 6.3.1.1 节对不同层次的多租户隔离机制进行介绍。

XaaS 应用往往对外以服务的方式提供其内部功能。传统的 Web 服务只面向单租户,那么,如何在很少或不进行代码更改的情况下使面向单个租户的 Web 服务支持多租户?如何保障服务的质量?这些问题将在 6.3.1.2 节进行初步讨论。

此外,为了达到多租户的目标,在数据层,如何设计多租户数据模型,以使得多个租户的数据能够尽可能的共享用来存储数据的空间资源,也是 XaaS 应用支持多租户面临的特定问题,这些已经在本书第四章 4.3 节进行了介绍,本章就不再赘述了。

由于 XaaS 软件支持多租户和第三方运营的特征，又引申出来资源共享和优化(optimizable)、可伸缩(scalable)、可用性(availability)、可靠性(dependability)和可配置(configurable)等其他几个基本特征。

首先，为了追求更低的成本和更高的利润，同一个物理平台或应用要服务于尽可能多的租户，租户之间资源要尽可能在物理上集中共享，并保障租户使用资源的效率。在 XaaS 模式下，应用软件统一部署在服务器端（往往涉及到多个数据中心），服务器端统一对其计算、存储和带宽资源在多个应用程序间共享和优化调度，并和最终用户之间可达成细粒度地服务质量保障协议。为了达到这个目的，需要先对 XaaS 模式下与资源共享和优化调度相关的实体进行合理的建模，然后利用合适的控制策略，实现资源的共享与优化。因此，本章将在下面的 6.3.2 节从多租户资源的抽象和控制两个方面来介绍多租户的资源共享与优化问题。

其次，在 XaaS 模式下，由于每个租户都可能拥有大量的潜在最终用户，因此在设计开发中要把实现可伸缩性作为重要的目标，应用软件应根据其实际负载的需求在线扩展。可伸缩性是分布式计算的经典问题，已经有一些成功应用的技术，这些技术在 XaaS 模式下往往同样适用。本章将在下面的 6.3.3 节对可伸缩技术进行扼要介绍。

再其次，由于 XaaS 模式下，用户使用但不拥有软件，软件实际运行在第三方运营中心，较传统用户掌握软件拥有权的情况，用户更加担心第三方运营中心一旦出现不可事先预见的故障将会对其业务造成无法弥补的损失。因此，XaaS 模式下，系统可用性和可靠性的问题就变得更加突出。本章将在下面的 6.3.4 节对可用性和可靠性技术进行介绍。

最后，由于 XaaS 模式下共享同一个平台的多个租户具有不同的应用需求，因此 XaaS 应用需要支持租户通过配置灵活且方便地进行应用定制。这些要求为传统分布式计算带来了一些的新问题。在 XaaS 模式下，租户往往没有对应用程序的构成组件直接进行修改的权限，而且 XaaS 模式下租户是通过因特网进行应用定制的，因此，传统技术无法直接适应 XaaS 模式的要求。本章将在下面的 6.3.5 节对 XaaS 模式的配置问题进行介绍。

6.3.1 多租户

我们先介绍一个实例，然后以这个实例为基础，来理解多租户的概念。该实例以一个物流报关领域的业务需求为应用场景，这个应用场景主要包括从工厂或货运代理公司到海关申报系统之间的业务流程。

A 是一家笔记本电脑的出口加工企业，员工规模 300 人。B 是一家国际货运代理公司，员工只有 10 人。ISV-1 则是一家独立软件提供商，该公司开发运营物

流系统，该系统可以支持公司 A 和公司 B 两个租户，共 310 名左右的最终用户来使用物流系统。它将允许公司 A 和公司 B 的租户管理员登录物流系统，对其各自的数据模型、业务逻辑和用户界面等进行定制，并且，公司 A 和公司 B 之间的业务互不干扰。

6.3.1.1 多租户隔离机制

保证多个租户数据和程序的隔离是多租户的基本问题。以上述场景为例，只有 ISV-1 开发的物流程序要被尽可能多的租户共享，才能够提高资源使用的效率，从而确保 XaaS 运营商的利益最大化。多个租户共享同样的资源，就提出了多租户隔离的问题：如何确保多个租户的数据和业务逻辑不互相混淆？如何确保一个租户的用户在未经过授权的情况下，不允许访问其他租户的数据和业务逻辑？如何确保多租户程序的使用者在使用 XaaS 服务时，完全感受不到其他租户的存在，如同自己在独享资源一样？

要解决这样的问题，首先，必须区分不同租户的数据和业务逻辑。可以将不同租户的数据和业务逻辑分配在不同的物理机器、同一台机器但不同的操作系统以及同一操作系统但不同的地址空间中，也可以在同一操作系统的进程中，对不同租户的数据和业务进行逻辑标识加以区分。

因此，XaaS 模式互联网分布式系统对多租户的支持可以实现在不同的层次上，采取不同的多租户隔离方法。例如，在中间件层和系统软件层不对租户进行区分，只在应用层针对不同的租户有不同的应用实例；在中间件层就开始支持多租户，不同的租户有逻辑上独立的中间件；或者在系统软件层，采用虚拟机技术，来为每个租户提供一个独立运行的虚拟操作系统环境。

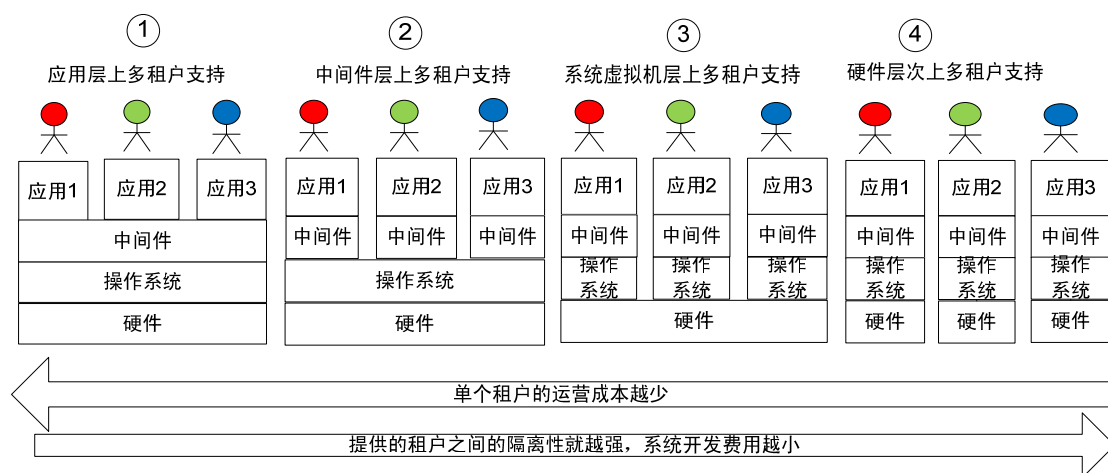


图 6.6 按照层次多租户的几种实现方式

1. 在应用层实现对多租户的支持

在应用层实现对多租户的支持，不同的租户共享服务器、操作系统以及中间

件，但仍然可以有多种不同的实现方式。不同的租户既可以共享应用程序的单一实例，也可以使用应用程序的不同实例。

在共享同一应用程序实例时，通常的实现方法是使用租户标识参数对应用程序的单一实例进行参数化。例如，如果应用程序有 Web 服务接口和实现，那么在接口中的操作和数据对象中添加租户 ID 参数。如果应用程序使用数据库表，那么可以在每个数据库表中添加一个表示租户 ID 的新列（当然，还可以有其他做法，第四章我们对多租户数据模型进行了详细分析和介绍）。不同的租户，可能根据租户 ID 具有不同的配置元素，例如，虚拟门户为每个租户提供不同的外观等。

对于不同租户共享服务器、操作系统以及中间件，但具有不同应用程序实例的情况，系统对不同的租户都创建一个单独的应用程序拷贝，并部署到应用服务器的共享实例中，在数据库层，通常的做法是复制应用程序的数据库表。与租户相关的所有数据和应用程序的定制信息都添加到相应的租户相关的应用程序拷贝或数据库表拷贝中。

这些不同的实现方式，根据它们对多租户特性从基本的功能支持到提供更高的资源共享程度可划分为四个不同的级别，如图 6.7 所示。这四种不同的级别也被称为“SaaS 成熟度模型”，最先由 Frederick Chong 和 Gianpaolo Carraro(Chong, Carraro, 2006)提出。

(1) 第 1 级（每个客户一个实例）：每次新增一个客户，都会新增软件的一个实例。

(2) 第 2 级（所有客户同一个版本，不同实例）：所有客户都运行在软件的同一个版本上，而且任何的定制化都通过修改配置来实现。

(3) 第 3 级（单实例、多租户）：所有的客户都在软件的同一个版本的同一个实例上运行。该级别下软件提供商部署一个应用的实例即可满足多个客户的需求。

(4) 第 4 级（单实例、多租户、可伸缩）：在第 3 级的基础上具备可伸缩性，可以通过硬件水平扩展（scale out）的方式来进行扩充。

删除的内容: 图

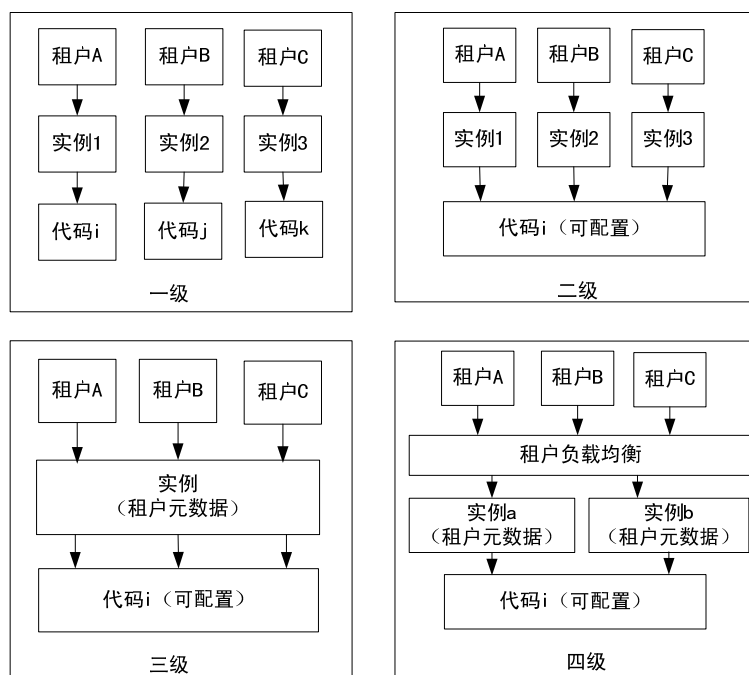


图 6.7 应用层多租户实现的成熟度模型

目前人们所通常认为的可以称为成熟的 SaaS 模式至少应该到达第 3 级，即支持单实例和多租户，Salesforce.com 就是一个“单实例、多租户”的典型例子。

这里，业务逻辑尤其是复杂的业务流程的隔离与共享，是 XaaS 模式面临的主要挑战之一，下面，给出了对业务流程进行租户隔离的三种方式：

(1) 不同租户使用不同的工作流引擎，这种情况下，有可能一个租户使用一个工作流引擎，也有可能一个租户同时使用多个工作流引擎。在这种方式下，不同租户的流程控制数据和工作流相关数据等存储到不同的数据库中。这种方式下，租户的迁移较为简单，不需对现有工作流进行改动。当一个租户可能同时使用多个工作流引擎时，需要利用传统分布式工作流的调度技术，对引擎资源进行合理分配。当多个租户的多个工作流引擎共享同一个服务器上的资源时，也需要对不同租户所使用的资源进行监控，并合理分配。这种方式下不同租户的流程数据隔离性较好。

(2) 多个租户共享同一个工作流引擎，但各自有不同的流程定义。这种方式下，需要使用不同的租户标识来区分不同的流程定义，因此，传统工作流的元模型需要进行相应的修改。这种方式下，不同租户的流程控制数据和工作流相关数据等可以选择存储到不同的数据库中，也可以采用共享数据库的方式进行。前者的优点是隔离性好，后者的优点是共享的程度更高，能够支持更多的租户。

(3) 多个租户共享同一个工作流引擎，并共享同一个流程定义。这种方式下，需要使用不同的租户标识来区分不同的流程实例，因此，必须对现有工作流引擎的实现进行修改。由于不同的租户可能具有不同的组织模型，对于工作流人工活

动的指派来说，也需要与不同租户的组织数据关联。这种方式下，不同租户的流程控制数据和工作流相关数据等可以选择存储到不同的数据库中，也可以采用共享数据库的方式进行。前者的优点是隔离性好，后者的优点是共享的程度更高，能够支持更多的租户。

下面，我们着重分析一下多个租户共享同一个工作流引擎，但各自有不同流程定义情况下的多租户支撑实现机制。首先，给出下面的支持多租户的工作流元模型。

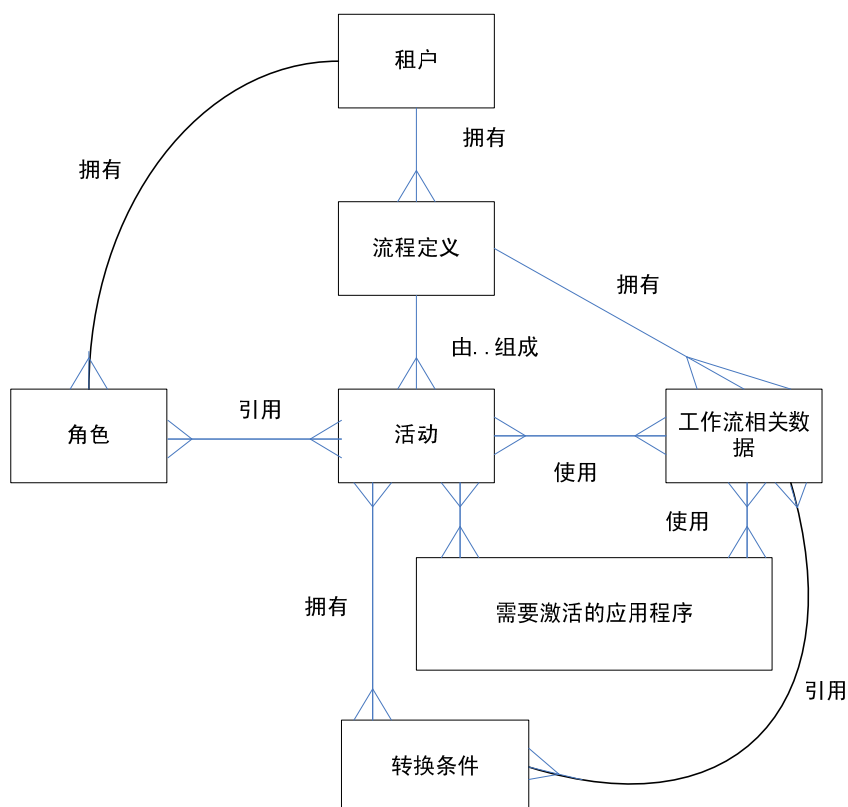


图 6.8 支持多租户的工作流元模型

与传统的工作流模型相比，在支持多租户的情况下，增加了“租户”这一实体。租户拥有多个流程定义，有自己的组织模型。不同的租户，在其组织模型中，可能拥有不同的角色。

与传统的工作流元模型相比，在支持多租户的情况下，流程定义除了包含诸如流程名称、流程 ID、流程启动和终止的条件、系统安全以及监控和控制信息等基本属性之外，还包括一种重要的基本属性——所属租户的 ID。这个属性反映了该流程定义是由哪个租户来定义、其相关的活动以及工作流相关数据等都隶属于哪个租户。

工作流通常分为两个阶段：构造时和运行时。在构造时，不同的租户进行各自的流程定义；在运行时，根据各租户的流程定义创建的流程实例也分属于各租

户。

2. 在中间件层实现对多租户的支持

在这种实现方式中，租户共享操作系统和服务器，不同的租户使用应用程序的不同实例，但是这些实例部署在中间件的不同实例中。由于中间件实例是不同的，所以每个租户有自己的操作系统进程（地址空间）。这种方法在相同的物理服务器上支持的租户数量比在应用层实现多租户的支持要少。

在图 6.9 的示例中，每个租户（公司 A 和 B）运行自己的应用程序物理拷贝（分别为 App1 和 App2），这些拷贝部署在自己的中间件物理拷贝中（分别为 M1 和 M2）。注意，租户可在不同的操作系统上运行，例如，在此示例中，A 在 Windows 上运行应用程序，而租户 B 在 Linux 上运行应用程序。与前面介绍的方法相比，这种方法需要对现有应用程序做的修改较少，这有助于加快部署速度。

删除的内容: 图

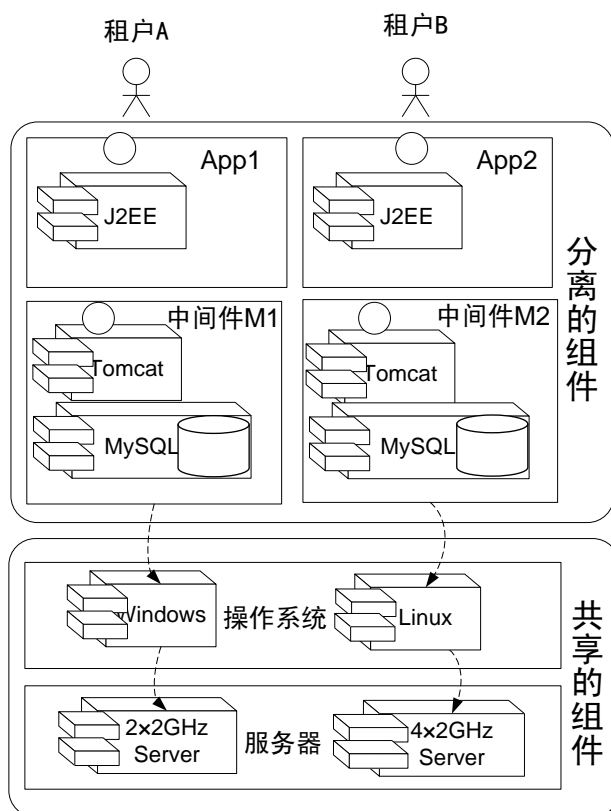


图 6.9 在中间件层实现对多租户支持的示例

3. 系统虚拟机层次上的多租户支持

虚拟机，即为由虚拟机软件模拟出来的计算机或称为逻辑上的计算机。虚拟机是支持多操作系统并行运行在单个物理服务器上的一种系统。因此，虚拟机是一套计算机软件，其基本功能是在一个操作系统里面模拟安装了另一个操作系统的计算机。习惯上称被模拟的操作系统为客系统(Guest OS)，运行虚拟机软件的

操作系统为宿主系统(Host OS)。采用虚拟机能够提供更加有效的底层硬件使用,因为每个虚拟机都可模拟与物理计算机相同的运行环境,包括硬件层、驱动接口、操作系统及应用层;同一物理计算机的多个虚拟机还可以相互连接起来形成网络,实现集群作用。

虚拟机根据它们的运用和与直接机器的相关性分为两大类。系统虚拟机提供一个可以运行完整操作系统的完整的系统平台。相反,程序虚拟机为运行单个计算机程序设计,这意味它支持单个进程。虚拟机的一个本质特点是运行在虚拟机上的软件被局限在虚拟机提供的资源里——它不能超出虚拟世界。

在系统虚拟机层次上支持多租户,多个租户共享物理服务器,但使用不同的虚拟映像以及不同的应用程序、中间件和操作系统实例。使用这种方法,在物理服务器(主系统)上安装服务器虚拟化之后,对于每个租户,服务供应商实例化一个虚拟服务器(客系统),它包含与这个租户相关的软件,包括中间件和应用程序。因此,使用这种方法,不需要为实现多租户进行大量支持多租户的应用程序或者中间件的代码开发。

图 6.10 给出一个采用这种方法的示例,其中在物理服务器上安装了本机系统管理程序(例如 Xen)。在这个示例中,底部的物理服务器 ServerA 有两个频率为 2Ghz 的 CPU,它被分为两个虚拟服务器(vServer1 和 vServer2),每个虚拟服务器各有一个 2 Ghz 的 CPU。虚拟服务器 vServer3 包含整个服务器 ServerB,有 4 个 2Ghz CPU。虚拟服务器还共享物理服务器的其他资源,比如内存、磁盘空间和网络连接。应用程序 App3 和 App4 部署在 vServer1 和 vServer2 中,为两个租户服务,App5 部署在 vServer3 中。

删除的内容:图

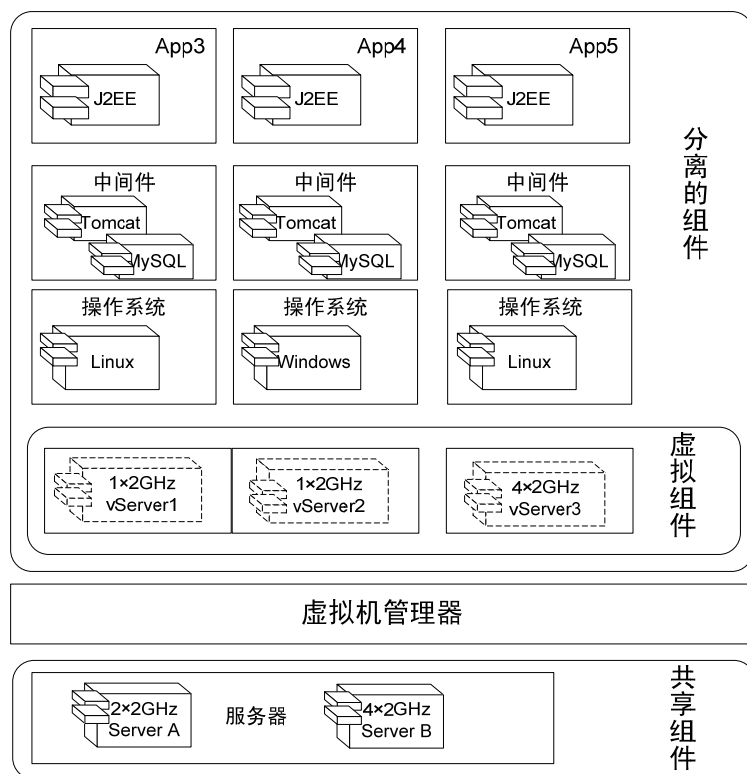


图 6.10 在虚拟机层实现对多租户支持的示例

4. 服务器层次上的多租户支持

在硬件层次上提供对多租户的支持，多个租户只共享数据中心的物理基础设施（比如供电和制冷），但是使用不同的物理服务器、操作系统、中间件和应用程序的不同实例。在图 6.11 所示的示例中，租户 A、B 和 C 使用三个不同的应用程序实例 AppA、AppB 和 AppC，它们在与租户相关的中间件实例、操作系统实例和物理服务器上运行。在以上介绍的四种方法中，这种方法提供最高的隔离性。

删除的内容: 图

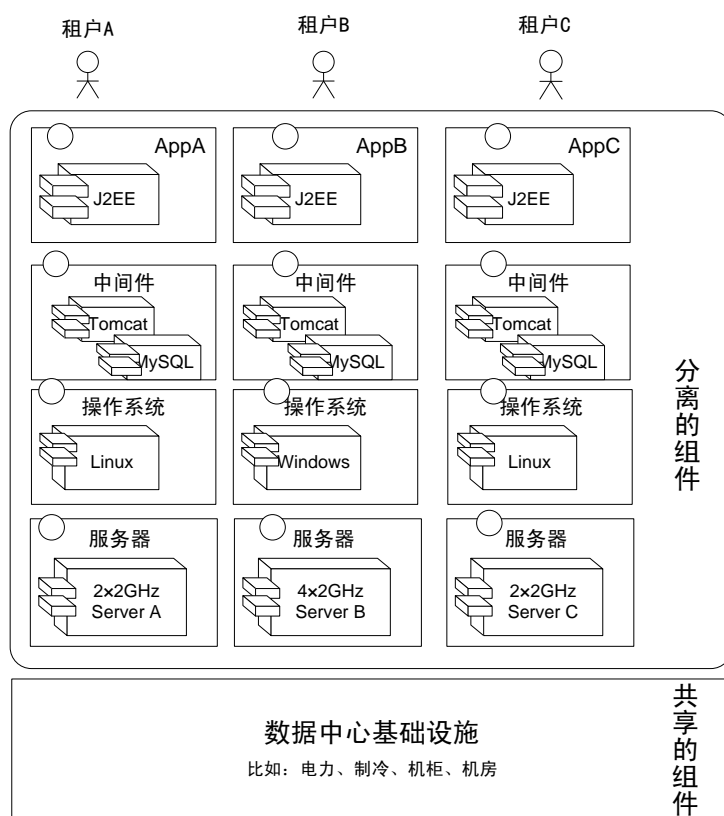


图 6.11 在硬件层实现对多租户支持的示例

比较不同层次上实现多租户的支持，有几个规律：(1) 实现多租户的层次越低，那么，其能够提供的租户之间的隔离性就越强。例如，在应用层实现多租户的支持，不同租户可以有自己的应用程序实例，但是它们却共享同一个操作系统进程，但在中间件层实现多租户的支持，每个租户具有自己的操作系统进程（地址空间），因此，在中间件层实现多租户的支持，其能够提供的租户之间的隔离性更强，但是，这种方式在操作系统和服务器层仍然有隔离问题，例如一个租户的用户有可能占用物理服务器中的所有 CPU 和内存。(2) 实现多租户的层次越低，那么，相同的硬件资源能够支持的租户的数量越少，单个租户的运营成本越高。(3) 实现多租户的层次越低，那么，所需要的系统开发费用越小。例如，对于在硬件层次上实现多租户的情况，不需要针对多租户功能进行额外的系统开发，而在应用层上实现多租户的情况，必须在应用层基于现有的数据库和中间件软件进行针对多租户功能的系统设计和开发。

6.3.1.2 支持多租户的服务提供机制

传统的 Web 服务只面向单租户，使现有面向单租户的 Web 服务支持多租户，面临如下挑战性问题：如何在很少或不进行代码更改的情况下使面向单个租户的 Web 服务支持多租户？例如，在不对 Web 服务的接口实现进行代码更改的情况下，如何使面向单个租户的信用审核服务支持多租户？

对 Web 服务支持多租户的问题，在不改变现有 Web 服务的接口实现的条件
下，需要增加一个中介层，对不同的租户的服务的请求和响应进行区分和隔离，
对不同租户的服务调用情况进行监控和管理。

首先，中介层需要能区分不同租户的服务的请求和响应。例如，面向单个租
户的信用审核服务接口为 `Check(input1, input2, ...)`，那么在中介层，需要实现一
个支持面向多租户的 Web 服务接口 `Check-Multi-Tenant(tenantID, input1,
input2, ...)`，中介层需要来标识不同租户的服务请求，并将不同租户调用 Web
服务的结果进行正确的标识后，返回给正确的租户。

在很多时候，往往需要针对不同的租户指定服务提供商，因此中介层还需要
对来自不同租户的服务请求按照业务规则进行正确的路由。例如，有两个服务提
供商都提供信用审核服务，它们分别是 ISV-1 和 ISV-2，这时，有两个租户 A 和
租户 B，其中，ISV-1 是租户 A 的指定服务提供商，而 ISV-2 是租户 B 的指定服
务提供商。那么，中介层就需要负责将来自租户 A 的服务请求路由到 ISV-1 提供
的服务上，而来自租户 B 的服务请求正确路由到 ISV-2 提供的服务上。

在多租户的情况下，对 Web 服务以及前文介绍的 Open API 的监控和管理更
加紧迫。这是因为，每个租户所拥有的最终用户的数量常常是无法预先估计的，
如果不加监控、提前进行约束或采取相关措施，一旦服务请求负载超过了现有资
源的上限，所有租户的服务使用都将受到影响。因此，中介层的另外一个重要职
责就是对多个租户的服务调用情况进行监控和管理。常见的监控和管理措施包
括：对来自不同租户 Web 服务或 Open API 的调用进行计数、对来自不同 IP 的
最终用户的服务调用进行计数以及对服务请求中的身份许可标识（常称为服务调
用的 Key）进行认证等。通过限制某 IP 或某租户在一段时间内服务调用总的次
数进行限制或者审核服务请求是否合法等方法来验证服务请求者的合法性，避免
用户负载无限制增长的现象发生。

以 Google Maps 为例，用户若想使用 Google Maps 所提供的 Open API 开发
应用，就必须使用注册域名来获得密钥，这样申请得到的密钥将对该域、其子域、
这些域中主机上的所有网址以及这些主机上的所有端口有效。Google 使用这种
方法来对其 API 的使用进行一定程度上的控制。API 的提供商也可以轻易地做到
通过限制来自某密钥的 API 访问次数来控制用户请求的负载。但由于 Google 基
础设施在可伸缩性保障方面的充分能力，目前，Google 对于每天可使用地图 API
生成的地图页面展示的次数是没有限制的。但是如果一个使用 Google Maps API
的网站每天有超过 50 万个页面展示还是需要向 Google 申请额外的处理流量。

6.3.2 多租户的资源共享和优化

在第三方运营过程中，多租户的资源共享和优化是一件十分困难的问题。由于 XaaS 运营中心运行的软件越来越复杂，而且软件的负载随时间动态变化，因此要在每个租户都达到一定 SLA 的条件下最大化资源的利用率是一件极具挑战性的事情。

在 XaaS 模式下，多租户的资源管理需要涉及三个实体：资源、资源消费者和资源管理者，并且三个实体之间存在控制关系。因此，只有先对这些实体进行合理的建模，才能基于实体间的控制关系，利用合适的控制策略，实现资源的共享与优化。下面就从多租户资源管理的实体抽象和控制过程两个方面来介绍多租户的资源共享与优化问题。

6.3.2.1 多租户资源管理的实体抽象

在多租户的资源管理中，物理资源包括 CPU、内存、磁盘存储、磁盘或网络 I/O 等。为了方便这些资源的管理，操作系统实际上为用户建立了这些资源的抽象，例如，操作系统通过线程管理对 CPU 资源进行分配，通过虚拟内存管理对物理内存进行分配，而通过文件的管理对磁盘存储进行分配。因此，操作系统将这些资源分配给租户时，租户是 CPU 时间和物理内存等资源的消费者，但是，站在操作系统内核的角度，线程及虚拟内存等却是 CPU 时间和物理内存等资源的消费者。

资源的下述属性会对资源管理的策略和机制造成影响：(1) 资源的独占或共享使用方式。当资源以独占方式使用时，在特定的时间内只能分配给一个资源消费者使用。(2) 资源的状态，是有状态的还是无状态的。当资源有状态时，这个状态通常与正在使用它的资源消费者相关。资源的状态在资源分配时创建，之后清除或存储起来以便资源重新分配时再使用。(3) 资源是以单实例的形式存在还是作为资源池的一部分存在的。典型的资源池例如内存页面、多处理器中的 CPU 资源或同构集群中的计算机资源。

在多租户的资源管理中，资源管理者可以抽象为服务器进程或者被动对象 (passive object) 的形式。对于前者，资源请求是以消息的形式发送。对于后者，资源请求是进程调用或方法调用的形式，这个调用可以是隐藏的。一个资源管理器可以是分布式的，它可以由一组使用相同协议的管理器组成，对用户只提供一个应用程序接口。

在多租户的资源管理中，资源消费者的抽象包括两种形式：

1. 资源容器

在单独的机器上，例如 Web 或数据服务器，资源消费者经常抽象为资源容器的形式，这里资源容器是包含一个应用完成特定独立活动所使用到的所有系统资源的抽象实体。对 CPU 时间来说，一个资源容器动态绑定到一些线程，每个线程为多个资源容器服务，并且绑定关系随着时间而改变。资源容器根据某资源分配策略得到 CPU 时间。资源容器也可以层次的方式进行组织，在最高层次上定义的资源容器根据全局的资源管理策略获取系统资源。

2. 集群区

在集群上，资源消费者往往抽象为集群区(cluster reserves)的形式，它实际上是资源容器的扩展，本质上是集群范围内所有节点上资源容器的聚集。分配给一个集群区的全局资源可以进一步动态分解到不同节点上的资源容器中。资源在多个资源容器之间进行划分的问题可以抽象为约束优化问题，其目标是在满足如下约束的条件下求解资源在每个节点上的分配：每个集群区上的资源分配应该与全局资源分配策略偏离最小；每个节点上资源分配的总和不能超出现有的资源数量；资源不能分配给不会使用它的容器（资源不能浪费）等。集群中具有一个资源管理节点，它按需或者周期性地求解资源优化方案，并将结果通知给每个节点上的本地资源管理器。

6.3.2.2 多租户资源管理的控制过程

和其他的系统管理一样，多租户的资源管理也可以看成一个控制过程。关于控制原理这里做简要介绍。对于一个随时间演化并且受外界扰动的系统，控制的目标是指能够按照预定的策略控制系统的行为。这里需要假设系统提供了一个执行器使得通过执行器的命令能够影响系统的行为。这样的控制方法主要包括开环控制（前馈控制）和闭环控制（反馈控制）两种(Hellerstein, Parekh, 2004):

1. 开环控制

开环控制是最简单的一种控制方式，是指受控客体不对控制主体产生反作用的控制过程，也即不存在反馈回路的控制。在这种控制中，控制系统的输出仅由输入来确定。在实际中则表现为控制主体在发出控制指令后，不再参照受控客体的实际情况重新调整自己的指令。

开环控制的原理是：在对系统情况和外界干扰有了大致分析研究的基础上，通过控制初始条件，使系统能不受外界干扰的影响准确无误地转移到目标状态。这种控制如 [图 6.12](#) 表示：

删除的内容: 图

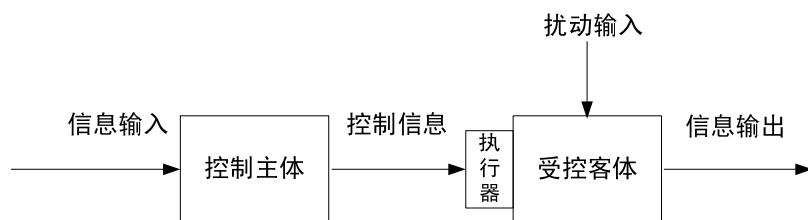


图 6.12 开环控制的原理

在管理中采用开环控制具有作用时间短及控制成本低等优点，在外界干扰较小且变化不大的情况下，有一定的控制作用。但由于没有反馈机制，开环控制无法发现和纠正计划和决策实施中与预定目标之间的偏差，缺乏抗干扰能力。因此，开环控制仅适用于那些受干扰影响不大且有规则的变化组织活动。在复杂多变的情况下，开环控制往往不能起到有效控制作用，有很大的局限性。

2. 闭环控制

闭环控制是根据控制对象输出反馈来进行校正的控制方式，它是在测量出实际与计划发生偏差时，按定额或标准来进行纠正的。在控制论中，闭环通常指输出端通过“旁链”方式回馈到输入。输出端回馈到输入端并参与对输出端再控制，这才是闭环控制的目的，这种目的是通过反馈来实现的。正反馈和负反馈是闭环控制常见的两种基本形式。其中正反馈和负反馈从达到目的的角度讲具有相同的意义。从反馈实现的具体方式来看，正反馈和负反馈属于代数或者算术意义上的“加减”反馈方式，即输出量回馈到输入端后，与输入量进行加减的运算后，作为新的控制输出，去进一步控制输出量。实际上，输出量对输入量的回馈远不止这些方式。这表现为：在运算上，不止于加减运算，还包括更广域的数学运算。

闭环控制的原理是：当受控客体受干扰的影响，其实现状态与期望状态出现偏差时，控制主体将根据这种偏差发出新的指令，以纠正偏差，抵消干扰的作用。在闭环控制中，由于控制主体能根据反馈信息发现和纠正受控客体运行的偏差，所以有较强的抗干扰能力，能进行有效的控制，从而保证预定目标的实现。管理中所实行的控制大多是闭环控制，所用的控制原理主要是反馈原理。这种控制如

图 6.13 所示：

删除的内容：图

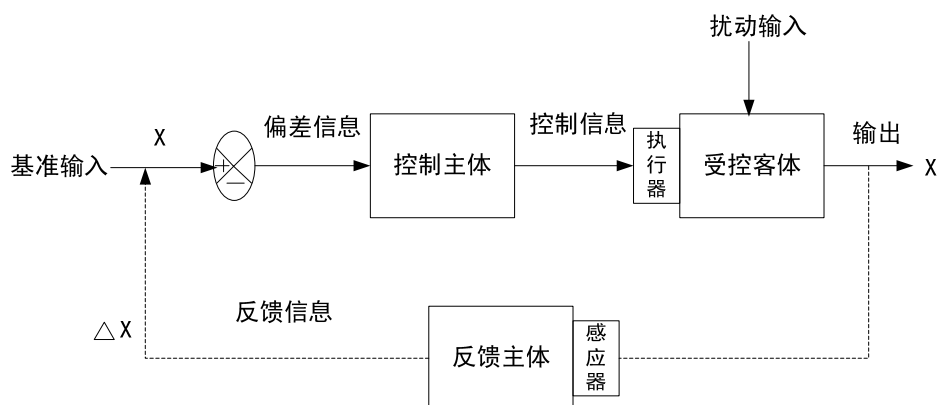


图 6.13 闭环控制的原理

闭环控制的优点是充分发挥了反馈的重要作用，排除了难以预料或不确定的因素，使校正行动更准确，更有力。但它缺乏开环控制的那种预防性，例如：闭环控制在控制过程中造成不利的后果才采取纠正措施。

如果一个系统处于这样的状态，资源管理者使用已经得到的资源能够达到资源消费者预定义的服务质量指标，那么该状态可以称为健康状态。资源管理者可看作一个控制系统，它运行一个控制算法，算法的目的是通过需求推断、优化部署、负载预测、动态资源提供以及应用监管等各种手段将系统维持在一个健康”的状态，以实现资源优化。这些手段可以基于上述开环、闭环或两者结合的方式来控制。下面首先介绍资源管理者的基本手段。

(1) 需求推断。XaaS 运营中心必须精确地推断出不同租户的资源需求，过高的估计资源的需求将浪费资源，过低的估计则可能导致租户所接受的服务质量受到影响。一般可基于应用的分析模型进行需求推断或者根据经验观察来进行推断。

(2) 优化部署。即当一个新的租户程序上线运行时，为其分配最合理的资源以便最大化整体的资源利用率。

(3) 负载预测。由于 XaaS 软件的用户负载是随时间动态变化的，因此，对用户负载进行预测就变得尤为重要，可根据预测结果合理调控资源的分配，从而在一段时间内达到整体最优。

(4) 动态资源提供。静态资源提供又称为资源预留，是根据租户对资源的需求的静态预计值来进行资源分配。动态资源提供是指根据动态变化的负载预测结果，动态地将资源提供给不同的租户。

(5) 应用监管。应用监管是为了保护应用不受突发负载的影响。为了保证现有服务的正常水平，在突发负载的情况下，应用监管允许运营中心自动忽略一些超出正常负载的服务请求。同样，在突发负载的情况下，对于一些优先级别较高的服务请求，应用监管也应该提供优先为其提供服务的机制。

在对上述手段的控制上，这里主要介绍基于开环控制的资源优化策略和基于

闭环控制的资源优化策略，其中基于开环控制的资源优化策略主要包括以下两种：

(1) 资源预留策略。这种情况假设资源消费者所需的资源是预先可知的，资源提供者分别为不同的资源消费者预留一定的资源，以保证其 SLA。这种方法的前提是资源的总量足够满足所有消费者的资源需求，在资源需求前后变化很大的情况下，这种方法会不可避免导致资源浪费的发生。

(2) 比例调度策略。按照一定的权重（比例）对一组需要调度的资源进行调度，让它们得到的资源与它们的权重完全成正比。比例调度策略可以采取的方法有轮转法、公平共享、公平队列和彩票调度法等几种，这些均是从传统操作系统进程管理中借鉴得来的经典方法。在基于集群的互联网分布式系统中，轮转法或者其变种如动态加权轮转法是经常被用来在不同的节点之间进行负载均衡的方法。关于互联网分布式系统中所采用的负载均衡算法，感兴趣的读者可进一步参考相关综述(Cardellini, Casalicchio, Colajanni, et al., 2002)。

为了能够应用基于闭环控制的资源优化策略，我们需要指定系统的表示模型、观察和控制的变量、使控制器能够与这些变量进行交互的感应器和执行器的手段以及控制器的控制算法。下面分别来介绍这些内容。

对一个计算系统的资源管理和性能进行建模有四种主要的方法方法(Sacha, 2007):

(1) 经验方法，这是最常用的一种方法。例如，早期为避免虚拟内存颠簸现象发生，系统被简单地建模为一个在三个状态（正常、过载和低负荷）之间根据资源占有率进行转换的状态转移图，其目的是通过调节允许共享内存的任务数目避免系统进入过载状态。该模型非常简单，但也有可能是非常有效的。

(2) 将控制系统看作一个黑盒，其内部组织结构对外不可见，只通过一系列的感应器和执行器与外部世界通信。假定系统按照一定的规律运行，且是线性和不随时间变化的。

定义 6.7 所谓线形系统是指状态变量和输出变量对于所有可能的输入变量和初始状态都满足叠加原理的系统。

定义 6.8 叠加原理是指：如果系统相应于任意两种输入和初始状态($u_1(t), x_{01}$)和($u_2(t), x_{02}$)时的状态和输出分别为($x_1(t), y_1(t)$)和($x_2(t), y_2(t)$)，则当输入和初始状态为 ($C_1u_1(t) + C_2u_2(t), C_1x_{01} + C_2x_{02}$) 时，系统的状态和输出必为 ($C_1x_1(t) + C_2x_2(t), C_1y_1(t) + C_2y_2(t)$)，其中 x 表示状态， y 表示输入， u 表示输出， C_1 和 C_2 为任意实数。那么，系统模型的参数值可通过执行一系列的实验（例如，周期性地向系统提交一些输入，并测量其输出响应）来求取。[Lath, 2000]

(3) 基于排队理论的方法(Chen, Iyer, Liu, et al., 2007; Doyle, Chase, Asad, et

al., 2003)。将系统抽象为一个排队网络，即一系列互相连接的队列，每个队列都与特定的资源或资源集合关联。

(4) 基于分析模型(Urgaonkar, Pacifici, Shenoy, et al., 2005)来表示系统行为特性的方法。通过分析系统具有的一些行为特征来进行建模，从而使分析后的模型能够模拟实际系统的行为特征。

由于资源管理的目标是提高 QoS，使用 QoS 相关的参数作为观察的变量是很自然的事情，例如响应时间，吞吐量等等。然而，这些变量的值并不容易测量和获取。因此，这里使用更容易获得的与资源使用相关的变量。资源使用相关变量包括 CPU 负载、内存占用率、电量消耗、I/O 通道速度以及集群中服务器使用的百分比等。

感应器主要任务是观察变量，执行器主要任务是控制这些变量。执行器采取的手段有如下几种：分配或释放一个资源分配单元，例如，CPU 的时间片、服务器集群中的一个服务器或通信信道中的一部分共享带宽；改变资源的状态，例如关闭服务器及修改 CPU 的频率等；从资源池中增加或减少资源；允许或拒绝资源消费者来竞争资源。

前面已经介绍了系统的表示模型、观察和控制的变量以及感应器和执行器的手段，下面简要介绍控制算法。目前使用较多的控制算法是基于比例或者积分的控制算法，这些算法假设受控客体是线形或者分段线形。基于微分的控制算法很少使用，因为它们容易对突然的负载变化作出很强烈的反映，但这种现象在互联网服务中是很常见的事情。对于一些更加复杂的情况，(Hellerstein, Parekh, 2004)等人正在研究相关算法。

在互联网环境下，系统往往呈现明显的非线性行为，再加上互联网分布式系统的用户请求负载较难准确建模，除此之外，感应器和执行器本身也是一个非常复杂的系统，它们自身的动态性也需要在整个控制系统中考虑。这就使得互联网分布式系统的多租户资源优化问题迄今为止仍然是一个开放的研究问题。

除上述内容之外，还有一个重要的问题是多租户环境下的安全和信任问题，即如何在多租户之间共享应用程序资源，以便只有属于租户 A 的最终用户可以访问属于该租户的实例，而只有属于租户 B 的最终用户可以访问属于该租户的实例？我们已经在第四章对多租户共享数据库的问题进行了专门的介绍，而多租户环境下的安全和信任问题我们将在第七章进行详细解释。

6.3.3 可伸缩性

对互联网环境下大规模的分布式系统来说，由于它们往往要支撑成千上万用户的并发请求，所以良好的可伸缩性是至关重要的特征之一。本书在第二章便对

互联网分布式系统的可伸缩性进行了概况的介绍。对于 XaaS 模式下的互联网分布式系统来说,所服务的用户往往无法事先预计,为了最大限度的为租户提供优质的服务,进而最大化其商业利益, XaaS 模式的互联网分布式系统的可伸缩性就显得尤为重要。传统分布式系统的可伸缩性的保障技术同样可适用于 XaaS 模式的互联网分布式系统中,下面我们从可伸缩性的度量、评价及其一般技术两个方面进行介绍。

6.3.3.1 可伸缩性的度量与评价

可伸缩性在并行计算领域已经发展了一系列的度量指标,这些度量假设程序在给定 k 个处理器上运行,并使用任务的完成时间 T 来测量系统的性能。并行计算机可伸缩性度量指标主要有三个(Grama, Gupta, Kumar, 1993): (1). 并行加速比(*Speedup*) S 即 k 个处理器与单位处理器的性能比率。理想情况下,加速比应该与处理器的个数呈线性关系 $S(k)=k$ 。(2). 并行效率(*Efficiency*) E 使用单位处理器的加速比 $E(k)=S(k)/k$ 来测量,理想情况下, E 值应为 1。(3). 从规模 k_1 到规模 k_2 的并行可伸缩性(*Scalability*)使用其效率的比值 $\psi(k_1, k_2) = E(k_2) / E(k_1)$ 来测量,理想情况下,其值为 1。此外,固定大小加速比(*fixed size speedup*)用来表示同样比例扩展基数的情况下,任务完成时间的比率: $S(k) = T(1) / T(k)$ 。还有一些附加指标,例如等效率(*isoefficiency*)和等速率(*isospeed*) (Hwang, Xu, 1998)等,这些指标用来在不同的情况下根据不同的约束条件来描述系统的可伸缩性。

然而,对互联网分布式系统来说,可伸缩性的度量并不能完全照搬并行计算中的上述度量指标,这是因为:

(1) 在互联网分布式系统中,系统的大小不能仅根据处理器的数量来度量,还必须考虑复制、不同类型和价格的网络和存储设备等因素,可以说,系统的大小是一个多维的度量属性。

(2) 在互联网分布式系统中,对系统性能的评价除任务的完成时间之外,更重要的是服务质量,指分布式系统提供服务的好坏,例如可用性的度量。

(3) 在互联网分布式系统中,处理器的成本、存储和带宽的成本及管理维护费用的成本等的计算都必须考虑在内,并且,成本的计算与采用软件即服务“pay-as-you-go”应用运营和使用模式相关。

(4) 对互联网分布式系统中,对系统进行扩展可能采取的方法并非简单的增加处理器、带宽和存储这样简单,还包括更加复杂和丰富的方法,例如复制和改变通信策略等,因此,在对系统的可伸缩性进行度量时,还需要将其采取的可伸缩性策略考虑在内。

Jogalekar 和 Woodside 提出了一种更加具有通用性的分布式系统可伸缩性度

量指标(Jogalekar, Woodside, 2000):

设 $\lambda(k)$ 为在问题规模 k 的情况下系统的吞吐量, 根据用户请求的每秒平均响应数来计算;

设 $f(k)$ 为每个请求响应的服务质量平均取值, 这个取值根据问题规模 k 情况下的响应的服务质量来计算得到;

设 $C(k)$ 为问题规模 k 情况下与每秒的吞吐量 λ 对应的成本 (Cost);

那么 $F(k) = \lambda(k) \cdot f(k) / C(k)$ 为系统每秒的生产率;

定义问题规模 k_2 与 k_1 时生产率的比值为分布式系统从规模 k_1 到规模 k_2 的可伸缩性:

$$\psi(k_1, k_2) = (F(k_2)) / (F(k_1)) \quad (6.1)$$

如果一个分布式系统在问题规模变化时, 生产率 F 随成本 C 的增加而增加或者随 C 的减小而减小, 即 ψ 始终保持大于 1 或者接近 1, 则可以说这个系统是可伸缩的。在具体的工作中, 可以为 ψ 值定义一个阈值, 例如定义当 $\psi > 0.8$ 时, 则系统是可伸缩的。在某成本固定的情况下, ψ 取阈值时的 k 值为系统在此情况下能够支撑的问题规模极限。

在上述求系统可伸缩性的参数中, 吞吐量 λ 的语义不言自明。成本 C 需要特殊说明一下, 它不是指一次投资成本, 而是每个单元时间所消耗的资源租赁成本, 它可以将处理器、存储、网络、软件及管理维护等各种成本计算在内。 $f(k)$ 是根据对系统的性能评价方法来决定的, 可以将响应时间、可用性及数据丢包率等各种与系统提供服务质量相关的因素考虑在内。例如, 假设我们只考虑系统的请求响应时间, 那么, 设 $T(k)$ 为问题规模 k 时的请求响应时间, 设系统的目标期望响应时间为 \hat{T} , 对请求响应的服务质量均值进行归一化, 归一化后的数值分布在 $[0,1]$ 区间内。因此, 定义 f 为:

$$f(k) = 1 / (1 + (T(k) / \hat{T})) \quad (6.2)$$

将其带入式(1)化简后则有:

$$\psi(k_1, k_2) = \frac{\lambda_2 \cdot C_1 \cdot (T_1 + \hat{T})}{\lambda_1 \cdot C_2 \cdot (T_2 + \hat{T})} \quad (6.3)$$

而作为一种特例, 并行计算系统的可伸缩性同样可以使用式(3)来度量。

式(3)是从理论上给出的系统可伸缩性封闭解, 但是在实践中, 其参数值一般并不容易获得, 为此, 下面我们介绍一种可伸缩性的数值求解方法(Jogalekar, Woodside, 2000)。

决定一个系统可伸缩性配置的变量集合可分为两部分: $(x(k), y(k))$, 其中, $x(k)$ 表示在给定问题规模 k 的情况下由系统选取的伸缩策略决定的参数集合; $y(k)$ 则表示可调参数集合, 通过调节 y 可以达到每个问题规模 k 时的最大生产率。

系统在每个 k 时 $(x(k), y(k))$ 的取值形成一条路径，可称为伸缩路径 (scale path)。例如，对于一个数据库系统来说，伸缩策略把请求用户个数、数据库大小和处理器个数都定义为 k 的函数，分别表示为

$$User(k)=k \tag{6.4}$$

$$Database(k)=10,000 \log_{10} k, \tag{6.5}$$

$$CPU(k)= \lceil k/100 \rceil \tag{6.6}$$

如图 6.14 显示 k 从 100 变化到 300 的过程中，在 (数据库大小, N 处理器) 二维空间的伸缩路径。对于给定的 k ，系统选取的伸缩策略以及可调参数的优化取值就决定了该数据库系统在 k 时的配置。

删除的内容: 图

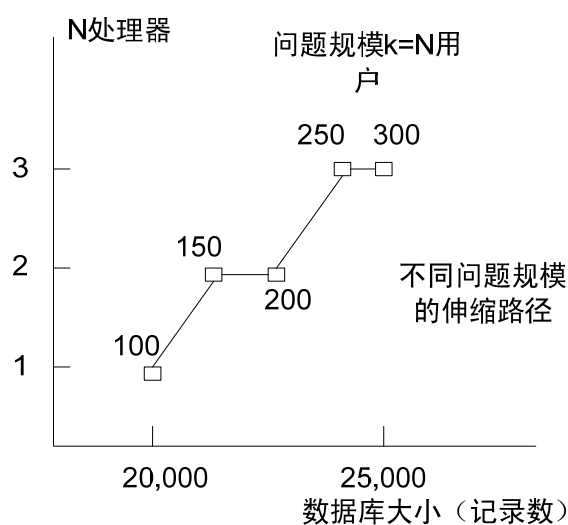


图 6.14 伸缩变量和伸缩路径

可伸缩性的数值求解首先对每个给定的 $x(k)$ 和 $y(k)$ 初始值，按照某合理的系统性能分析模型 (例如，在一些合理的假设条件下，很多分布式系统都可以采用排队网络模型对其性能进行建模) 求解响应时间和吞吐量的数值近似值，进而求得系统在每个 k 时的生产率，即系统初始伸缩路径；然后对当前伸缩路径上每一个问题规模 k ，利用数值搜索算法调节 y 参数，以使得当前问题规模 k 时的生产率逼近最大。若系统的初始伸缩路径有多个，则还需要在所有初始伸缩路径的结果中选择对应生产率最大的伸缩路径。

上述数值求解方法还是显得比较繁琐，相对来说，通过对系统资源的乐观假设来求取系统生产率的临界值则比较容易。这种假设省去了优化过程，在每个问题规模 k ，可以只计算一次。通过求解临界值同样可反映系统伸缩路径上各种资源消耗和系统开销等变化的趋势，如果临界上限值已经说明系统可伸缩性已经很差，则实际情况只会更差。因此，求解临界值能够在很快的时间内且较简便地为系统可伸缩性的设计带来很多有用的指导信息，Jogalekar 等人给出了求解临界值的相关算法(Jogalekar, Woodside, 2000)。

6.3.3.2 可伸缩性技术

可以按照应用程序的类型对伸缩技术进行分类。我们可以将应用程序按照其瓶颈所在分为事务密集型应用程序、数据密集型应用程序和网络 I/O 密集型应用程序三大类。其中，事务密集型应用程序通过数据库存储数据，其大部分的操作就是数据库记录的增删改查。由于这些操作一般通过事务实现，我们将它们称为事务密集型应用。这类应用程序的扩展技术是提高应用服务器和数据库的性能，或者将组件复制并分布到系统中的其他节点上，在实践中一般采用支持事务密集型的高吞吐量数据库，以及应用程序服务器集群。数据密集型应用程序以一定的格式存储大量的用户数据，并响应用户对数据的使用请求，在实践中一般采用高性能的分布式存储系统，利用数据的划分、复制和缓存技术，将数据分布到系统各处以提高性能。网络 I/O 密集型应用程序需要大量的网络 I/O 访问传递实时或非实时数据，例如基于 Web 的网络视频会议，用户之间需占用大量的带宽传递多媒体数据流，一些数据密集型应用在高访问量的工作条件下也可以视为网络 I/O 密集型应用，如 Web 邮件服务。这类应用程序的扩展技术是提高服务器的带宽，或者将网络 I/O 分布到系统中的其他节点上，在实践中，可以使用具有良好带宽的服务器或内容分发网络（content delivery network）技术来提高此类应用程序的扩展性。

上述三类应用程序的伸缩技术也无非采用了垂直扩展（scale vertically 或 scale up）技术和水平扩展（scale horizontally 或 scale out）技术两类。前者是指对分布式系统中的单个节点增加资源，例如，为一台服务器增加 CPU 和内存资源。由于垂直扩展技术增加了可供虚拟操作系统和虚拟应用模块等共享的资源，因此，它能够使得我们充分利用虚拟化技术所带来的好处。水平扩展则是指在一个分布式系统中增加更多的处理节点，例如，一个 Web 站点的服务器从一台增加为三台。这两大类方法各有优缺点，水平扩展会带来管理的复杂性，节点之间的 I/O 吞吐量和延迟问题也会给编程人员带来一些复杂因素。但相对来说，水平扩展的经济成本相对于垂直扩展低，因此水平扩展一直以来是人们最常用的扩展技术。近年来，随着虚拟化技术发展的成熟，一个虚拟系统能够快速简便地运行在 Hypervisor（虚拟机管理器）之上，而其系统购买和安装成本都要比购买一台真实的独立服务器并在其上安装的费用要小，因此人们对垂直扩展的需求也开始旺盛起来。

水平扩展技术又可分为两类：分布技术以及缓存和复制技术。分布技术是指把组件分割成多个部分，然后再将它们分散到系统中去。针对数据库，对数据表进行划分也是一种常见的提高数据库伸缩性的技术。复制技术是指对组件进行复制并将其拷贝分布到系统中去。缓存是复制的一种特殊形式，它往往指由客户发

起的资源拷贝。由于缓存和复制在系统的多个地方重复同一份数据，这就会引发不一致性问题。解决这类问题的关键是如何将更新立即同步到其他拷贝上，并保证并发操作在每个拷贝上具有相同的执行顺序。

在上述伸缩技术中，数据库的可伸缩性技术又是一个可以单独来讨论的话题，例如，一些研究者对关系数据库模型是否是实现分布式环境下可扩展数据管理的最佳模型提出了质疑。我们在第四章已经对此问题进行了详细讨论。

6.3.3.3 CAP定理

上面，我们介绍了可伸缩性的保障，但是在一个互联网分布式系统中，要同时保障系统的可伸缩性和一致性并不容易。美国加州大学伯克利分校的 Eric Brewer 教授发现，分布式系统的一致性(consistency, 所有用户看到一致的数据)、可用性(availability, 总能找到一个可用的数据副本)和分区容忍性(tolerance to network partition, 即使在网络发生分区的情况下, 仍然满足上述两点)三者是不可能同时实现的, 任何设计高明的分布式系统只能同时保障其中的两个性质, 这被称为“CAP定理”(Brewer, 2000)。Seth Gilbert 和 Nancy Lynch 等人于 2002 年在 ACM SIGACT 的论文中对上述 CAP 定理进行了形式化证明(Gilbert, Lynch, 2002)。CAP 定理的工程意义在于, 架构设计师不要把精力浪费在如何设计能满足三者的完美分布式系统, 而是应该进行取舍, 选取最适合应用需求的其中之一。

首先, 这里的一致性不同于数据库事务 ACID 性质中的一致性, 数据库 ACID 中的一致性是针对事务而言, 而这里的一致性是针对数据而言, 它保证在分布式系统的任意一组请求/响应操作下, 同一个数据在同一时刻只有一个值。

其次, 这里的可用性是指当系统一直可用时, 系统中每一个未失效的节点接受的请求都会返回响应。如果只从这方面考虑, 这对可用性的要求是比较弱的, 因为它并没有限制从请求到响应的需要多长时间; 但是如果和另外一个属性, 即和下面要介绍的分区容忍性结合起来之后, 这对可用性的要求就比较强了: 即使发生了严重的网络失效, 每个请求也都必须得到响应。

最后, 所谓分区容忍性, 是指网络允许丢失从一个节点到其他节点发送的消息, 除非全部网络失效, 否则不允许系统出现无效的响应, 所谓无效的响应是指响应操作不是原子性的(Lynch, 1996)。当网络被分区的时候, 从一个分区的节点发送到另外分区节点的所有消息将会丢失。这时, 一致性要求隐含着即使部分消息没有被发送到目的节点上, 每一个响应操作是原子性的。而可用性要求则隐含着即使发送的消息丢失, 每个接收到客户端请求的节点也都必须响应。

在互联网环境下, 应用的规模较传统应用有了很大的增长, 这时, CAP 定理就显得尤为重要。这是因为, 对事务数量规模较小的应用来说, 数据库在维护

ACID 性质时所导致的延迟对整个系统的性能以及用户的体验来说都不明显。但是，对于很多大规模的互联网应用来说，由维护 ACID 性质所导致的延迟会比较突出，从而可能对用户体验产生很大的影响。例如，在用户结算账单的过程中，填入详细的信用卡信息后，由于延迟，用户得到了一个“付账失败”的返回结果，此时，用户就会担心是否处于已经付款但购物失败的状态？对此，亚马逊公司曾经断言，每增加十分之一秒的额外延迟，都会使他们的销售额减少 1%。而 Google 公司也注意到，半秒钟的延迟增长都会使得他们的流量减少五分之一。在这种情况下，CAP 定理可以让人们确认，一致性和其他性质不可兼得，因而不必浪费精力在对一致性过于严格的要求上，而是可以通过放松对一致性的要求来减少延迟，通过对可用性和分区容忍性的保障来提高用户体验。

对 CAP 定理，Gilbert 和 Lynch 的论文中有形式化的证明。在这里，我们也可通过一个简单的实例来对 CAP 定理进行一个直观的解释。

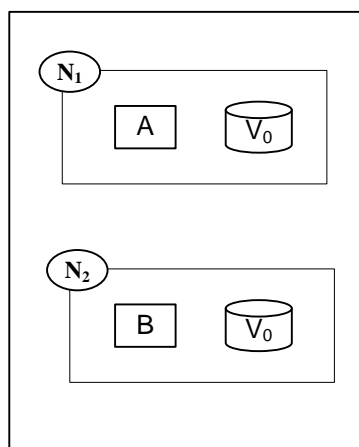


图 6.15 网络中的两个节点 N1 和 N2 共享同一数据项

图 6.15 中示出了网络中的两个节点 N1 和 N2，它们共享同一数据项 V，其值为 V0。在节点 N1 上执行的算法 A 可以认为是安全可靠的。同样，在节点 N2 上也执行类似的算法 B。在这个实验中，A 向数据项 V 写入新的值，而 B 读取 V 的值。正常情况下数据项的更新如图 6.16 所示：

删除的内容: 图

删除的内容: 图

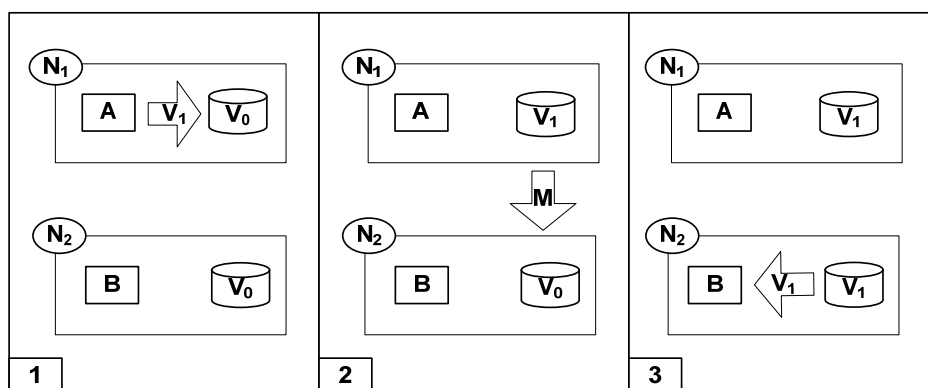


图 6.16 正常情况下数据项的更新

(1)A 向 V 写入新的值 V1; (2)更新节点 N2 中数据项 V 的消息 M 从节点 N1 传送到节点 N2; (3)此时, B 读到的值为更新后的值 V1。

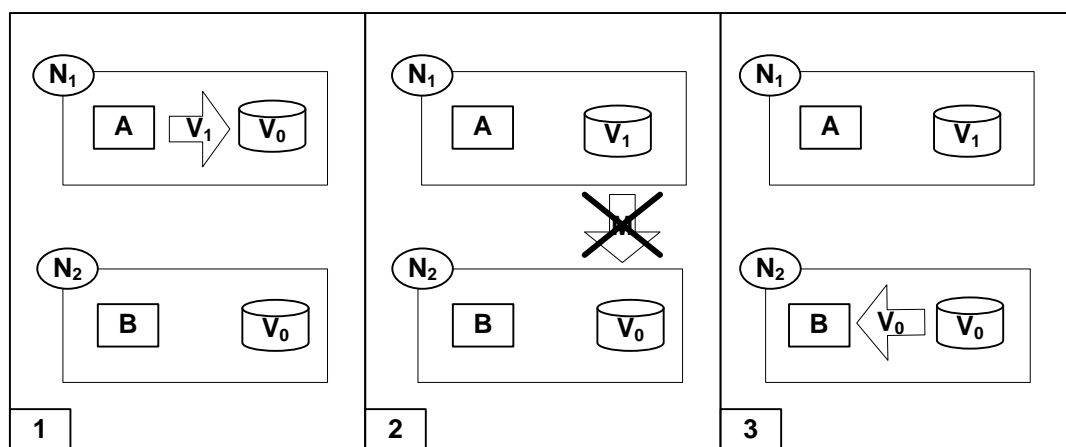


图 6.17 网络发生分区时数据项的更新

如果网络发生分区, 即从 N1 到 N2 的消息无法正确送达, 那么, 在步骤(3)发生时, N2 就包含了不一致的值, 如图 6.17 所示。

如果 M 是异步消息, N1 就无法知道 N2 是否收到了该消息。即使保证 M 可以送达 N2, N1 仍然无法获知消息是否由于分区时间或者 N2 的暂时失效而被延迟送达。即使在 M 为同步消息的情况下仍然于事无补。这是因为, 同步消息仍然将 A 在节点 N1 上的写操作, 以及从 N1 到 N2 的更新操作默认为一个原子操作, 上面的延迟仍可能发生。Gilbert 和 Lynch 的论文还证明了即使在部分同步模型(每个节点上都有顺序时钟)的情况下, 仍然无法保证上述操作的原子性。

因此, CAP 定理告诉我们, 要想使得 A 和 B 高度可用(例如, 都具有很小的延迟), 并且, 使得节点 N1 到 Nn 能够容忍网络分区(消息丢失和进程失效等)情况的发生, 有些情况下, 我们不得不容忍系统具有不一致的中间状态: 对一些节点来说, V 的值为 V0, 而对其他节点来说, V 的值为 V1。



图 6.18 从事务的角度理解 CAP 定理

若从事务的角度来分析, 如果有定义在数据项 V 上的事务 α , 那么, α_1 就是写操作, 而 α_2 为读操作。在一个本地系统中, 通过加锁来隔离 α_2 中

删除的内容: 图

在 $\alpha 1$ 结束之前的任何读操作，就可以获取系统的一致性。

但是，在分布式的系统中，还必须保证节点之间的同步消息正确完成。除非我们能够控制 $\alpha 2$ 发生的时间，我们就无法保证它一定会看到和 $\alpha 1$ 同样的数据项。然而，所有控制方法（阻塞、隔离和集中管理等）都要么会影响分区容忍性，要么会影响 $\alpha 1$ (A) 和/或 $\alpha 2$ (B) 的可用性。

6.3.4 可用性与可靠性

本书基础篇第二章已经介绍了分布式系统可用性和可靠性的基本概念。可靠性指标用来测量一个系统在没有故障和失败的情况下，能工作多长时间。而可用性是一个比值，是指一个系统正常运行时间的百分比。记系统正常运行直到系统失效的平均时间，即系统失效前的平均正常运行时间为 $MTTF$ (mean time to failure, 平均失效时间)，记用于修复系统和在修复后将它恢复到工作状态所用的平均时间为 $MTTR$ (mean time to repair, 平均修复时间)，那么，系统可靠性可定义为 $MTTF$ ，而系统的可用性可定义为：

$$Availability = \frac{MTTF}{MTTF + MTTR} \quad (6.7)$$

在 XaaS 模式下，对可用性的度量必须将多租户的因素考虑在内，假设 XaaS 模式下互联网分布式系统共有 N 个租户，当故障发生时，平均有 X 个租户受到影响，那么，系统可用性的度量公式即为：

$$Availability = 1 - \frac{MTTR}{MTTF + MTTR} \times \frac{X}{N} \quad (6.8)$$

当 $MTTR$ 、 $MTTF$ 和 N 为常量的时候， X 就成为影响整个系统可用性的关键因素。换句话说，面向租户进行故障隔离的目标是使得 X/N 的值，即故障在多个租户之间的传播比率，尽可能小。

我们经常看到一些互联网服务运营厂商宣称其系统的可用性达到“五个 9”（即 $Availability=99.999\%$ ）甚至更高的可用性，这个数据同上式的道理一样，说明了一个系统平均一段时间内（通常是一年左右）的失效间隔时间。但是，对于这个测量方法和指标，有几点是需要注意的：(1) 这个数据是个平均数，在可用性不能达到 100% 的前提条件下，任何系统在任何时候，都还是有可能发生失效，对租户来说，如果发生失效的时间正好是租户运营关键业务的热点时间，那么对其造成的影响同样是巨大的；(2) 系统的可用性并不等同于用户最终得到的可用性。用户最终得到的可用性是一个整体的概念，假如一个系统有 5 个组件构成，如果其中 4 个组件的可用性都达到了五个 9，但另外一个组件的可用性比较低，那么整体的可用性还是大打折扣。再比如一个系统的可用性达到了五个 9，但是由于第三方网络的失效，还是无法保障用户最终得到的可用性达到五个 9。(3) 可

用性数据的测量并不容易，方法不适当可能会导致较大误差。有些实际系统通过监控和系统日志的方法来计算可用性，但是，监控的间隔时间（例如是五分钟一次，还是三分钟一次）是 >0 的，间隔时间内的失效是系统监控的盲点，因此，如果间隔时间选择不适当，系统可用性与实际情况的误差就会很大。

鉴于上述原因，将可用性定义为设计目标而非实际目标更为合理。这里，定义高可用性是指设计时所确定的可用性级别，要求其满足或超出其应用要求。

为了更好地了解故障对系统的影响，并决定采取相关的措施，人们对失效模式进行了总结和分类。从失效严重程度不同的角度可将失效分为下述几种类型 (Van, Tanenbaum, 2007):

- (1) 崩溃性失效，是指服务器停机；
- (2) 遗漏性失效，是指服务器不能响应到来的请求，包括不能接收到来的消息和不能发送消息；
- (3) 定时失效，是指服务器的响应在指定的时间间隔之外；
- (4) 响应失效，是指服务器的响应不正确，这种不正确的响应可能是响应的值错误，也可能是服务器偏离了正确的控制流；
- (5) 随意性失效，又称为 **Byzantine**（拜占庭）失效，是指服务器可能在随意的时间产生随意的响应。
- (6) 也可以从故障来源的角度，将故障分为下述几种类型：
 - (7) 硬件故障，例如 CPU、内存、存储设备及网卡等硬件发生的故障；
 - (8) 环境和物理故障，指导致计算机系统发生故障的外部物理故障，例如电源故障等；
 - (9) 网络故障，例如错误的路由信息及地址解析错误等，造成了包的丢失和损坏等；
 - (10) 数据库故障，指数据库系统发生的故障，造成数据库故障的原因有很多，应用程序的挂起、数据库索引的错误、资源不足等都可能是引发数据库故障的来源；
 - (11) 中间件故障，指中间件发生的故障；
 - (12) 应用程序和 Web 服务故障，指应用程序或 Web 服务发生的故障，造成应用程序和 Web 服务发生故障的原因更为复杂。

提高系统可用性的手段可从两个方面考虑：如何增加系统的 **MTTF**，如何减少系统的 **MTTR**，也就是延长系统正常运行的平均时间和减少系统的恢复平均时间。提高系统可靠性和可用性的关键技术主要包括下述几个方面：

1. 隔离与冗余技术。

隔离技术使得一个发生故障的组件之间不互相影响，最简单的例子如利用

Xen 虚拟机技术，当虚拟机失效而重启时，并不影响同一台物理机器上在其他虚拟机上运行的程序。

当系统的一个组件发生故障时，利用冗余技术可以使得系统由另外一个具有同样或相似功能的组件继续提供服务。例如，在一个面向服务的分布式系统中，可以通过提供冗余服务的方法，来提高服务的可靠性和可用性。对一个开放式的系统来说，也可以通过发现和组织互联网上的第三方服务来扩大获得冗余服务的机会。但是，开放的第三方服务并不能保障与原有的服务兼容，这种冲突可能来自于其他管理域对服务的契约型要求，也可能是因为新的服务可能需要经过转换或者与其他服务组合后才能达到与原有服务相同的功能。因此，需要对这些服务进行分析、评估、组织和转换，从而能够在服务发生故障时，找到能够替代该服务的其他服务提供给用户使用。

根据所使用的资源不同，冗余技术可分为时间冗余和空间冗余两类。其中，时间冗余是指重复执行某个操作来提供冗余信息，如使用 Redo 操作和消息超时重发机制等。空间冗余又有硬件冗余和软件冗余两种，硬件冗余是指使用额外的 CPU 和总线等硬件来提供冗余信息，软件冗余是指使用备用或独立版本的算法来提供冗余信息。冗余技术还可分为静态冗余和动态冗余两类，其中，静态冗余技术在给定的模块中采用冗余组件来掩盖硬件或软件故障，使得模块的输出不受故障的影响。动态冗余技术允许错误出现在模块的输出中，而是采取故障检测后进行恢复的机制来提升系统的可用性和可靠性。

2. 故障检测技术。

系统运行采用两个同样的组件，通过监控和比较这两个组件的差异就可以进行故障检测，这种方法能够检测出所有非重叠的单个故障，但是却无法对重叠发生在这两个组件上的故障进行检测，因为这时两个组件仍然没有差异。

失效检测器(failure detector, FD)是一种基于消息收发的超时机制来进行失效检测的组件，失效检测器与被检测对象之间通过发送周期性的(T_f)心跳消息来检测后者的存活性。根据消息交互策略的不同，失效检测的实现可以分为 PUSH 和 PULL 两种基本方式(Sacha, 2007)。

(1) PUSH 方式：每个被检测对象主动向它的检测者 FD 周期性的发送“心跳(heartbeat)”消息宣告其存活性，FD 被动地接受消息。如果 FD 在给定的时间段 T_w 内没有收到被检测者的“心跳”消息，则怀疑它发生了失效。PUSH 策略通常也称为“心跳”策略；

(2) PULL 方式：失效检测器 FD 周期性地向被检测对象发送存活询问消息，被检测者收到询问消息后，发送一个应答消息。如果 FD 在时间段 T_w 内没有收到被检测者的应答，则怀疑它发生失效。PULL 策略通常也称为 Ping 策略。

在要求相同的检测效果的前提下，PULL 策略发送的消息数约为 PUSH 策略的两倍，造成的网络负载较大；但是，PULL 策略是一种主动的检测方式，可以只在需要的时候才发起检测，检测结果具有更好的时效性。总之，这两种策略各有特点，系统实现时可以根据应用需求和运行环境的特点，选用合适的策略。

此外，系统也可以通过验证中间或最终结果是否在系统预设的合理范围内，或者验证是否将对象的访问限制在授权范围内等方法来检测系统故障。采用奇偶校验、循环冗余检测以及校验和等也是进行故障检测的常见方法。

为尽早发现并定位系统故障，XaaS 模式下每个租户都必须具有监控其运行实例状态的能力，并采取例如心跳检测等方法来将当前租户的运行状态以一定的时间间隔定时提交给系统。需要注意的是，XaaS 模式下一旦检测出某租户发生的故障，就要设法避免该故障传播到其他租户。一个基本的原则是出现故障的租户立刻释放该租户所占用的共享资源。

3. 失效恢复技术。

失效恢复是指系统用正确的状态取代错误的状态。失效恢复有两种，一种是回退恢复（backward recovery），指从当前的错误状态回到先前的某一个正确状态，另外一种则是前向恢复（forward recovery），当系统进入错误状态时，从可以继续执行的某点开始把系统带入一个正确的状态。回退恢复要求系统记录系统的历史正确状态，每次记录系统的当前状态时就称为一个检查点（check point），回退恢复是一种较为通用的恢复技术，但对系统来说会造成很大的开销。相比而言，前向恢复开销较小，但它的前提是必须预先知道系统下一步的新状态是什么，因此并非对所有的应用程序适用。

对于运维过程中的人为因素造成的错误，提供系统级的 Undo 补偿机制是十分必要的。系统级的 Undo 包括对软硬件升级和配置操作的补偿，也包括应用管理方面的操作，其难点在于操作恢复可能会引起的冲突如何解决。

此外，值得强调的是，在分布式系统中完成失效检测以及数据或进程的复制等功能，都依赖于可靠的通信机制和协议来支持。因此，可靠的客户-服务器通信和可靠的多播通信是在分布式系统中实现可用性和可靠性保障的基础。这往往也是从分布式系统学科的角度（例如，区别于从软件工程学科的角度）可用性和可靠性保障机制区别与其他学科问题最明显的地方。

6.3.5 可配置

在 SaaS 模式下，多个租户对数据库、业务逻辑和用户界面等都会提出自己的定制需求。例如，在不影响公司 A 数据模式定义的情况下，公司 B 如何将新的数据字段引入“合同订单”的共享数据库表？如何在不更改代码的情况下自定

义网站外观？例如，如何在公司 A 和公司 B 的“合同订单” Portlet 信息组件(一些可以聚合到门户页面的标记语言代码片段)中显示不同的字段？如何在不更改代码的情况下允许租户自定义业务逻辑？例如，公司 A 为中型公司，可以直接报关，而公司 B 的报关流程必须委托给其他相关单位进行？为了满足租户的这些要求，在 ISV-1 公司开发的运营物流系统中，描述每个租户的数据、业务逻辑和用户界面等的元数据，都需要专门地进行组织和管理。系统中的其他服务以及每个租户的客户端，可以通过对这些元数据的增加、删除、修改和查看来改变该租户对数据、业务逻辑以及用户界面等的定制。租户在使用物流系统的一段时间内，对其初始定制的可能还会有所变动。

互联网分布式系统实现 SaaS 模式下多租户定制的关键是多租户元数据服务。多租户元数据服务主要用于管理不同租户的应用配置信息，系统中的其他服务和客户端通过元数据服务来获取并更新每个租户的应用配置信息。在支持 SaaS 模式多租户定制的互联网分布式系统中，如图 6.19 所示，所有租户的数据和功能组件都不是静态的，而是在运行时由根据元数据服务所提供的应用配置信息动态生成相应的数据表和功能组件等。支持 SaaS 模式多租户定制的互联网分布式系统体系结构是元数据驱动的，其应用数据、描述不同租户应用配置信息的元数据以及租户的运行时引擎之间遵循“关注分离”原则，从而使得不同租户之间可以独立地完成运行时引擎的更新、应用的修改、配置等功能。

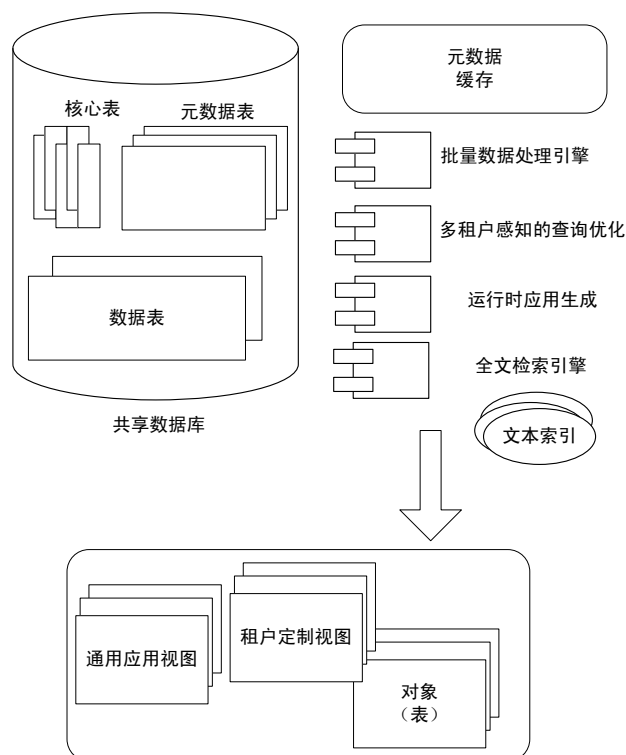


图 6.19 元数据驱动的支持 SaaS 模式多租户定制的互联网分布式系统体系结构
元数据服务所管理的元数据涉及如下几个方面：

删除的内容：图

(1) 数据模型的扩展配置信息。不同租户的数据模型存在差异，并且不是一成不变的。租户应可通过对数据模型的扩展配置来修改其数据模型。

(2) 用户界面配置信息。不同的租户可定制带有租户所属组织特点的个性化用户界面，包括字体和颜色等各方面。

(3) 工作流和业务规则配置信息。SaaS 模式的应用必须能够适应不同的租户在 workflow 方面的差异。例如，发票跟踪应用的一个客户需要一个经理对发票进行确认的活动；另外一个客户则需要发票必须经过两个经理的顺序确认方可进行下面的活动；而第三个客户虽然要求发票同时经过两个经理的确认，但却允许经理的确认可以并行进行。SaaS 模式的应用还必须能够支持租户对流程的配置，以使得应用的业务流程和租户所在组织的整体业务流程协调一致。

(4) 访问控制策略配置信息。每个租户都具有为其最终用户创建账户的责任，并决定最终用户的资源访问权限。用户的权限是根据一定的安全策略来跟踪管理的，租户也应通过对策略的配置来更改用户权限。

关于多租户数据库的定制，我们在第四章已经做了详细介绍。在 SaaS 模式下，不同的租户可以对其可访问的业务功能进行定制。业务功能的元数据主要描述了租户可以访问的功能集合包含哪些具体的功能。为此，系统首先需要将所有的业务功能进行分解，分解的单位称为“原子功能”，即具有独立功能和不可再细分的业务功能，例如，“订单创建”和“订单修改”等功能。原子功能之间有可能存在依赖的关系，例如，“查看客户产品列表”功能依赖于“查看产品列表”的功能，如果租户没有购买“查看产品列表”的功能，那么，也就不能对客户产品列表进行查看。因此，租户可定制业务功能的元数据主要包括下面几个方面的内容：对功能集合的定义（包括功能集合的名称、关键字和内容描述等）、对功能集合所包含的原子功能的定义（包括原子功能的名称、关键字和内容描述等），还包括原子功能所依赖的其他原子功能的定义。图 6.20 列出了一个多租户可定制功能的元数据模型。

删除的内容: 图

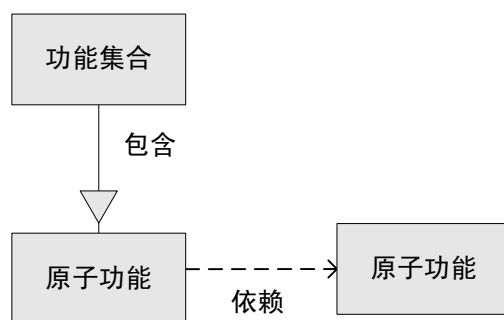


图 6.20 业务功能的元数据模型

用户界面的配置信息主要包括两个方面，一是用户界面包含哪些系统菜单；

二是租户的功能页面包含哪些内容。

菜单具有下述特点：每个租户都对对应一套属于自己的菜单；每一个菜单都与一个原子功能或者功能集合对应；菜单一般按照层次方式进行组织，同级菜单之间还有明显的顺序关系。因此，菜单的元数据就必须包括下述属性：菜单所属的租户、菜单所对应的功能、菜单的上级菜单以及菜单的标识和名字等基本属性。

而租户功能页面的元数据需要包括该页面上放置的页面元素、元素的位置、顺序以及元素的呈现名字和呈现颜色等基本特征。事实上，租户功能页面元数据是比较繁杂的，用户对功能页面的定制也是最常变化的。为了获取更好的灵活性，在实现时，往往将其描述成 XML 的形式。

在流程的可配置方面，SaaS 模式下流程的定制和传统工作流系统面临的问题并没有太大的区别。前面，我们根据多个租户是否共享同一个工作流引擎和是否共享流程定义等的不同，将 SaaS 模式下的多租户流程共享分为三种情况。根据这三种情况的不同，有不同的多租户流程隔离方案。

租户可定制的信息包括以上四种元数据，为了方便租户访问和修改其配置信息，我们可以将元数据组织成一个“配置项集合”，它将可配置信息分组聚集在一起。每个集合由多个配置项组成，每个配置项都是四种元数据中的一种。由于并不是每个租户都可以对所有的配置项进行访问和修改，因此，“配置项集合”还描述了租户访问配置项的授权信息。

“配置项集合”是从 SaaS 系统设计者的角度对可配置信息进行的分组。事实上，每个租户都可以根据自己的需求，进行四种元数据的配置。每个租户对元数据的配置并不影响上述“配置项集合”根据租户对配置项的访问权限对元数据进行的分组。为了支持每个租户对元数据的配置，可以定义属于每个租户的配置项集合，称为“租户配置项视图”。

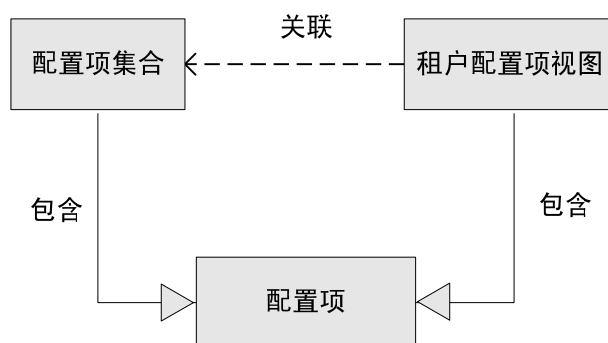


图 6.21 一种元数据的组织模型

为了向不同的租户提供灵活的可配置功能，也可以进一步将配置项集合组织成层次配置单元（scope）的形式。在顶层 scope 之下可以以任意的层次建立一个多个 scope。Scope 之间的关系来决定子节点如何继承并重载父节点的配置属性。

基于上面的元数据模型，SaaS 模式下的元数据服务需要提供一套统一的接口来操作这些元数据，包括配置项集合、配置项以及租户配置项视图的创建、修改、删除和查看等操作。由于 SaaS 模式对数据、界面、业务功能和流程以及最终用户的访问控制策略的定制是需要提供每个租户使用的，并且为了方便租户管理员进行定制，还需对不同的元数据提供相对统一的定制风格。因此，SaaS 模式下的元数据服务还必须提供一套可以直接使用的元数据控件，供业务模块开发时使用，以统一风格和降低实现可配置型的复杂度。

和传统的开发商定制应用不同，在大多数时候，在 SaaS 模式下对数据、界面、业务功能和流程以及最终用户的访问控制策略的定制工作都是由不具备专业 IT 知识背景的“租户管理员”以自助 (self-service) 的方式来进行的，他们面对如此繁复的配置选项，很容易产生错误的操作，并很可能有在多个配置项之间产生冲突的现象发生。因此，元数据配置功能的易用性就显得特别重要。

为了尽可能少的避免错误的发生和提高易用性，可以采用配置向导的做法。SaaS 系统的管理员预先为租户定义一系列的配置向导，每个向导都针对一类比较典型的租户需求。配置向导包括一些运行时的定制组件、帮助用户进行配置的用户界面和操作指南以及保证配置质量的配置验证规则。在“配置向导”的帮助下，“租户管理员”能够以自助的方式，简单的设置系统可配置点的值。定制的结果将存储在支持租户隔离的元数据库中。在系统运行时，将从元数据库中获取这些定制信息，并应用在租户的应用实例上。

Force.com(<http://www.salesforce.com/platform/>)是一个基于 SaaS 模式的支持多租户按需定制应用的平台，它支持两种方式的租户应用程序定制开发：一种是使用原始的平台应用开发框架；另外一种是使用平台提供的 API 进行定制开发。

平台应用开发框架为租户应用的定制开发者提供原始的用户界面，支持开发者进行诸如应用数据模型（包括定制对象及其字段和对象之间的约束关系等）的创建、安全和共享模型（用户和组织结构等）的创建、用户界面（界面元素的位置、数据实体表单和报表等）、以及表达业务逻辑的工作流程等。

使用平台应用开发框架提供的用户界面进行租户程序的配置，无需任何代码的编写工作。

Force.com 还提供原始的集成开发环境(IDE)，为租户程序的定制开发者提供访问平台内置功能的简单途径，开发者无需编写复杂和易错的代码就可以实现一些通用的应用功能。这些平台的内置功能包括：

(1) 声明性的工作流：为对象实例的插入或更新等操作预先定义一些触发活动，例如任务的执行、email 提醒、数据字段的更新或消息的发送等。

(2) 加密/屏蔽字段：开发者可简单的将某文本字段配置为对其内容进行加

密，并且在用户界面上显示时，使用输入屏蔽功能。

(3) 验证规则：支持开发者在不进行任何编码的情况下，强制实行领域完整性规则。

(4) 公式字段：开发者可较为容易的为数据对象增加一个统计或计算功能的字段。

(5) 聚集字段：开发者可创建跨多个数据对象的字段，在一个父对象中聚集子字段的信息。

Force.com 提供的另外一种进行租户程序定制开发的方式是基于 API 的方式，其 API 是与基于 SOAP 的开发环境兼容的。为了访问 Force.com 提供的 Web 服务，开发者首先下载一个 WSDL 文件，然后使用该文件生成一个 API，来访问相应的 Web 服务。

Force.com 提供两种类型的 WSDL 文件，一种是企业级的 WSDL 文件，它是对一个组织数据模型的强类型的表示，提供了对该组织的数据模式、数据类型和字段的描述信息，允许组织和 Force.com 的 Web 服务之间紧密集成。企业级 WSDL 随着该组织应用模式中定制对象或字段的添加和修改而相应的发生变化。Force.com 还提供另外一种 WSDL 文件供 Salesforce.com 的业务伙伴使用，它们可以为多个组织机构开发客户端应用程序。该 WSDL 文件只提供了对 Force.com 对象模型的松散表示，其目的仅仅是便于访问组织中的数据，而非进行紧密的集成。

Force.com 提供了一种强类型的和面向对象的过程化程序设计语言 APEX，定制开发者可以使用它来声明程序变量和常量，执行传统的流程控制语句（例如 if-else 和 loop 等）、进行数据处理操作（例如插入、更新和删除等）以及事务控制操作（例如 setSavepoint 和 rollback 等）。开发者可以编写 APEX 例程来为很多应用事件增加定制的业务逻辑，包括按钮点击、数据更新、Web 服务请求和定制的批处理服务等事件。

APEX 类似于 Java 语言，它是 Force.com 平台向用户交付可靠的多租户应用所必需的组件。例如，Force.com 自动在一个 APEX 类中自动对嵌入的 Sforce Object Query Language(SOQL)和 Sforce Object Search Language(SOSL)语句进行验证，来避免运行时错误的发生。平台为合法的 APEX 类维护相应的对象依赖信息，并使用这些信息避免那些导致依赖应用受到破坏的元数据变化发生。

APEX 语言还提供了对多租户资源共享和优化调度机制的支持。例如，Force.com 紧密地监控 APEX 脚本的执行，对其使用的 CPU 时间、内存和能够执行的查询数量等进行约束。这些约束对于保证多租户系统的整体可扩展性和性能来说是必要的。

为了进一步避免某些编写不合理的程序对平台整体性能的影响，Force.com 对一个新的租户应用的部署过程进行严格的管理。在租户将一个定制应用从开发状态升级为产品状态时，Force.com 要求对其 APEX 例程进行单元测试，并且，单元测试必须覆盖不少于 75% 以上的源代码。Force.com 在其沙箱环境中执行单元测试，在这个过程中，对该应用对平台整体是否造成性能的影响进行评估。

下面以一个例子来说明 Force.com 的配置过程。这个例子的目的是创建一个很简单的仓库管理系统，可以自底向上开始，首先创建一个跟踪货物的数据库模型，然后向这个数据库模型中增加业务逻辑，包括：用来保证足够存活的验证规则、当售出商品时更新存货总量的工作流规则以及对大宗发货单的电子邮件通知等。当数据模型以及相应的业务逻辑配置好之后，我们就可以创建相应的用户界面来显示产品的详细目录，并生成一个可对外发布的网站。

下面先来看一下创建简单仓库应用的过程。首先，创建一个定制对象-“Merchandise”，“定制对象”相当于关系数据库中的表格。图 6.22 展示了一个货物定制对象的主要 IDE 界面。“Merchandise”定制对象创建好之后，需要向其中增加一些字段，例如描述、价格和数量等。

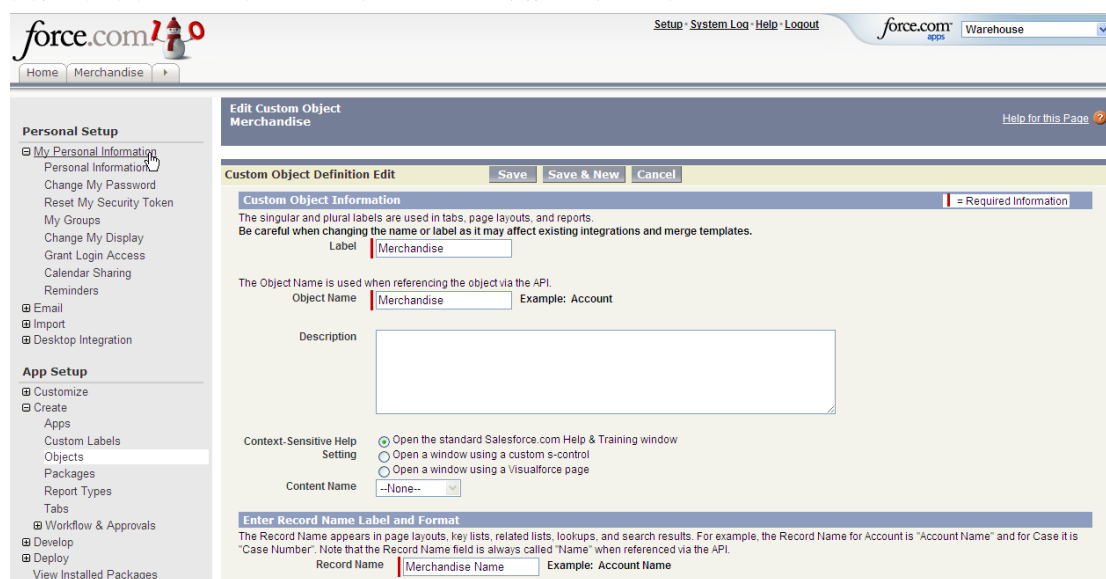


图 6.22 定制对象的创建界面

标签(Tab)是 Force.com 中用来发现和组织对象和记录的简便方法。Force.com 提供了为定制对象创建相应标签的便捷方法，当标签建立好之后，用户可以点击标签来创建、查看和编辑定制对象的记录。

在 Force.com 中，应用(Application)对应一组标签，图 6.23 展示了一个名字为 Warehouse 的应用并将 Merchandise 标签添加进去后的情形，点击 Merchandise 标签中的“New”按钮，可以向数据库添加 Merchandise 记录。

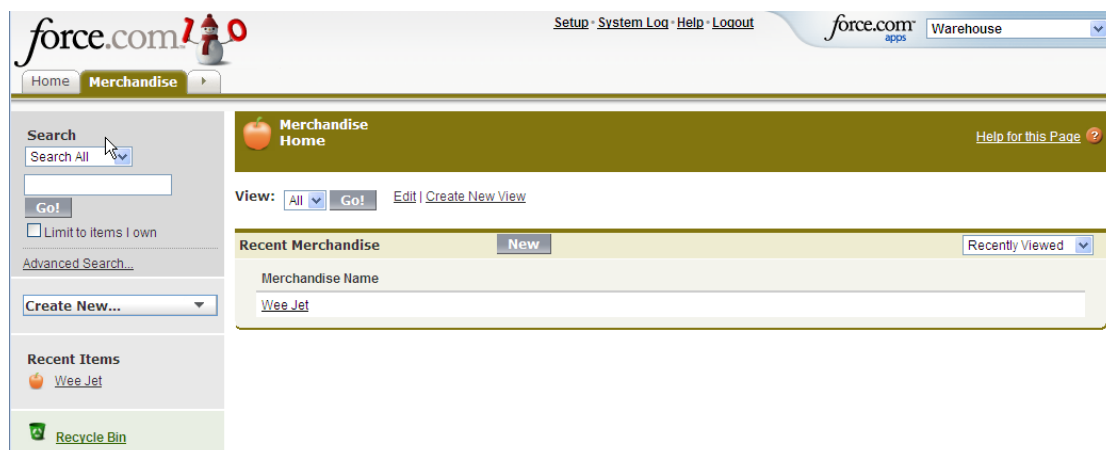


图 6.23 Warehouse 应用

接下来，创建一个“Invoice Statement”的定制对象，它具有一个“Status”字段和“Description”字段，再创建一个“Line Item”的定制对象，它具有一个“Unit Price”字段和“Unit Sold”字段。Invoice Statement 是由一组 Line Item 的记录构成的，而 Line Item 记录中 Unit Price 字段的值来自于 Merchandise。因此，接下来，在 Line Item 对象和 Invoice Statement 对象和 Merchandise 对象之间建立关系。

在 Force.com 中，是通过“关系”类型的字段来建立对象之间关系的。关系字段存储父记录的 ID，并将父记录和子记录的界面提供给用户。Force.com 定义了两种关系字段类型，一种是“Lookup Relationship”，该类型将一个对象“链接”到其他对象，从而使得用户可以从一个对象中查看另一个对象的记录；另一种是“Master-Detail Relationship”，该类型的字段将在子对象（detail）和父对象（master）之间建立关系，注意，关系字段对所有子记录都是必须的，并且，一旦关系字段的值被保存，就不能变化。父对象中的删除操作会级联影响到子对象。

“Lookup Relationship”能够用来创建一对一和一对多的关系，而“Master-Detail Relationship”在两个对象有紧密绑定的时候使用。例如，考虑“博客”和“博客帖子”，如果博客被删除，那么相应的博客帖子都需要被删除，这时就应该使用“Master-Detail Relationship”。在“Master-Detail Relationship”中，父对象可以包括一个“rollup summary”字段，这种字段保存子记录中的聚合值。例如，你可以使用这些字段来计算子记录的个数，一个子记录中某字段值的相加总和，或者计算子记录中某范围内某字段的最小值或最大值。

在这个例子中，在 Line Item 对象的详细内容界面中，创建一个数据类型为“Master-Detail Relationship”的字段，系统会提示用户填写“Related To”的值，用户若选择“Merchandise”，就创建了这两个对象之间的 Master-Detail 关系，其中，父对象为“Merchandise”，子对象是“Line Item”。用同样的方式来建立 Invoice Statement 对象和 Line Item 对象之间的 Master-Detail 关系。这些关系建好之后，

Force.com 就为用户生成了可增加、删除和修改相应 Line Item 记录的用户界面，图 6.24 是 Line Item 的一条记录，从图中可以看出，每条 Line Item 记录都指定了其父记录（Merchandise 和 Invoice Statement）的值。

The screenshot shows a 'Line Item Edit' form. At the top left is an orange icon and the text 'Line Item Edit 1'. Below this is a header bar with 'Line Item Edit' and three buttons: 'Save', 'Save & New', and 'Cancel'. The main section is titled 'Information' and contains the following fields:

Line Item Number	1
Unit Price	1.00
Units Sold	4
Merchandise	Wee Jet
Invoice_Statement	INV-0001

At the bottom of the form, there are three buttons: 'Save', 'Save & New', and 'Cancel'.

图 6.24 Line Item 的一条记录

图 6.25 示出了一个界面，可以较容易地为 Invoice Statement 对象创建一个 Roll-up Summary 类型的字段，来计算其子对象 Line Item 中 Value 字段各值相加的总和。

The screenshot shows the 'Step 3. Define the summary calculation' interface. It includes the following sections:

- Select Object to Summarize:** Master Object is 'Invoice_Statement', Summarized Object is 'Line Item'.
- Select Roll-Up Type:** Radio buttons for COUNT, SUM (selected), MIN, and MAX. A 'Field to Aggregate' dropdown is set to 'Value'.
- Filter Criteria:** Radio buttons for 'All records should be included in the calculation' (selected) and 'Only records meeting certain criteria should be included in the calculation'.

Buttons for 'Previous' and 'Next' are visible at the top and bottom right.

图 6.25 使用 Roll-up Summary 字段计算 Invoice Statement 的值

图 6.26 示出了为 Line Item 对象创建一个验证规则的界面，开发者可以在不进行任何编码的情况下，来验证用户的输入值是否合法，并给用户一定的提示信息。图中，开发者设定了这样一条验证规则：“Merchandise__r.Total_Inventory__c < Units_Sold__c”，即当售出的货物数量不足存货总量时，为用户报错。

Error Condition Formula ! = Required Information

Example: `Discount_Percent__c > 0.30` [More Examples ...](#)
 Display an error if Discount is more than 30%

If this formula expression is **true**, display the text defined in the Error Message area

`Merchandise__r.Total_Inventory__c < Units_Sold__c`

Functions

-- All Function Categories --

ABS
 AND
 BEGINS
 BLANKVALUE
 BR
 CASE

ABS(number)
 Returns the absolute value of a number, a number without its sign

[Help on this function](#)

Error Message

Example: `Discount percent cannot exceed 30%`

This message will appear when Error Condition formula is **true**

Error Message `You have ordered more items than we have in stock`

This error message can either appear at the top of the page or below a specific field on the page

Error Location | Top of Page Field `Units Sold`

图 6.26 创建验证规则

开发者可以创建一系列的工作流规则，例如，设定当添加新的 Line Item 对象时，其 Unit Price 字段自动从 Merchandise 的 Price 字段中读取。图 6.27 和图 6.28 分别示出了对这条规则的触发条件和相应活动的声明界面。相应的，在这个例子中，还需设定当创建或更新 Line Item 时，相应的 Merchandise 库存总量的值得到更新的规则。

删除的内容: 图

删除的内容: 图

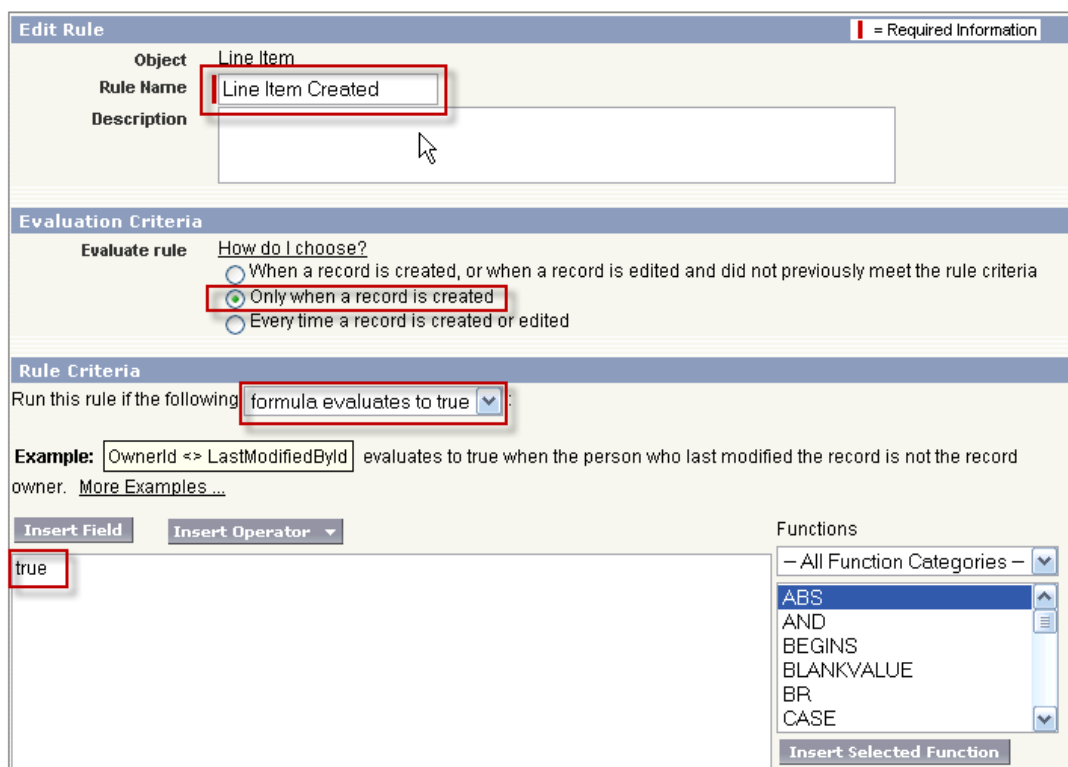


图 6.27 workflow规则的触发条件声明

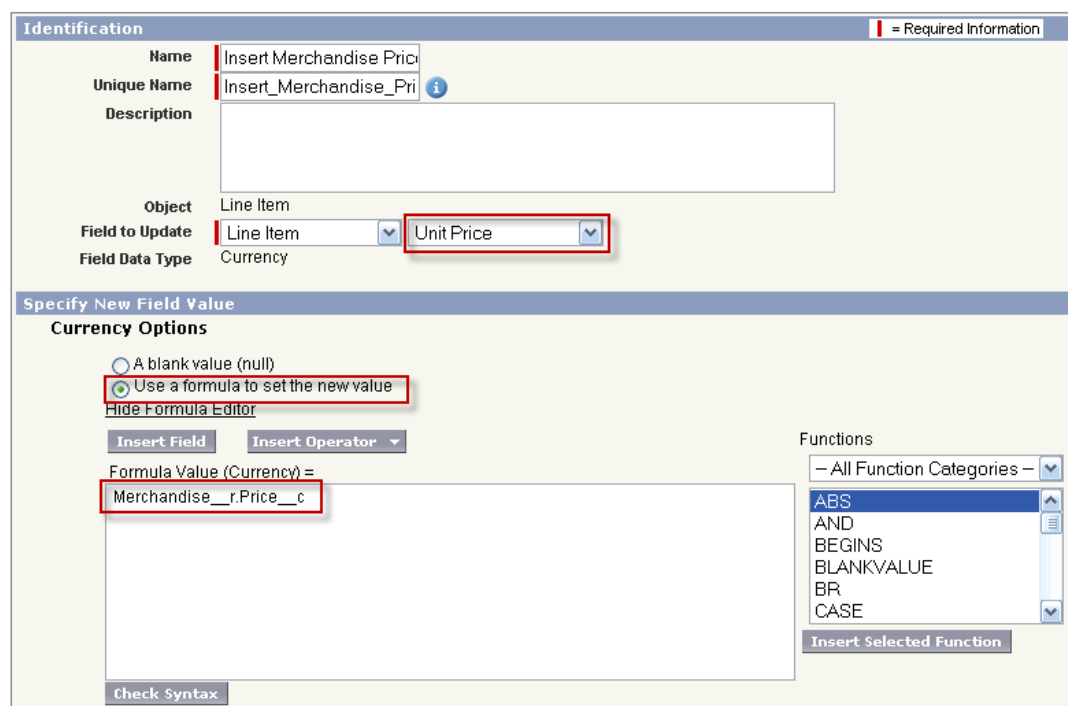


图 6.28 workflow规则的触发活动声明

开发者还可以通过 Force.com 很方便地设定这样的工作流规则，使得超过 2000 美元的发货单都必须经过经理的审批才能通过。为实现这样的规则，当发货单超过 2000 美元时，需要向经理发送 Email，经理收到后可以对其进行审批。首先，创建一个 Email 模板，如图 6.29 所示。然后，创建一个审批流程（图 6.30），

删除的内容：图
删除的内容：图

在“Approval Assignment Email Template”字段中，选择已经创建好的 Email 模板，在发送电子邮件的触发条件中，设定发货单超过 2000 美元的条件，然后配置电子邮件的发送对象为“经理”角色。

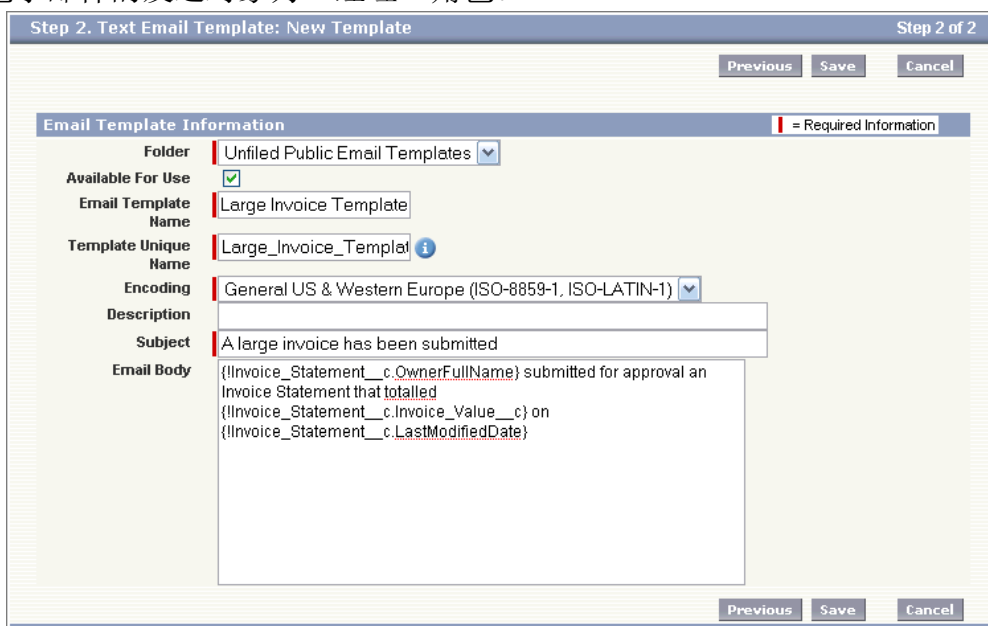


图 6.29 创建电子邮件模板

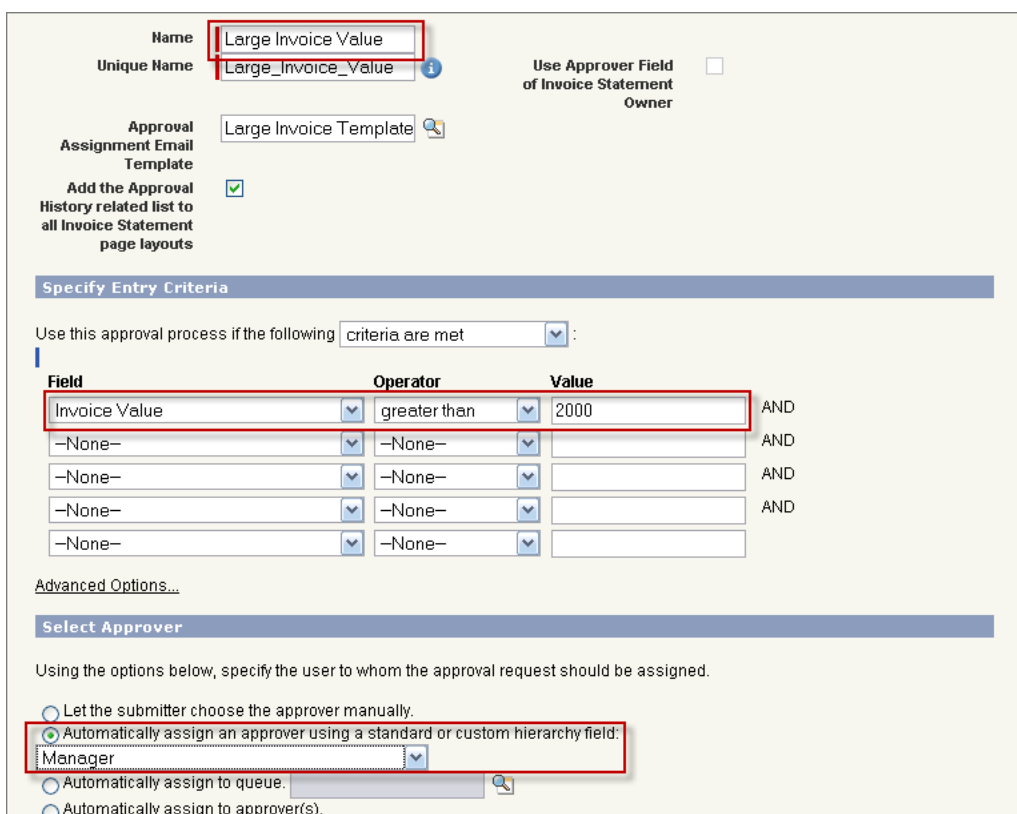


图 6.30 审批流程设定

开发这还可以通过 Fore.com 提供的 IDE，使用 APEX 语言来进行业务逻辑的编写。接下来，开发者可以来定制用户界面，并创建一个小型的带有域名的 Web 站点。然后，使用 Apex 语言，来创建一个简单的类及其方法，负责将货物

种类和数量呈现在网页上，并且，当用户点击“购买”时，能够调用后端的数据库，执行相应的数据库更新操作，并将操作结果呈现在用户界面上。这里，我们就不对这个过程进行详细的叙述了。

6.4 典型实例分析

Google AppEngine 是一个提供平台服务的 XaaS 环境，本章以它为背景，通过利用 AppEngine 开发和部署托管应用的基本方法和原理的介绍，对 AppEngine 可伸缩性的验证以及对 AppEngine 实现原理的简要分析，来介绍 XaaS 模式的第三方运营和优化的典型实例。

6.4.1 AppEngine 基本原理

Google AppEngine 提供了 Python 和 Java 两种应用程序开发环境，用户可以任意选择一种来开发自己的应用程序。这两种开发环境的功能都是类似的，本书将基于 Java 版本来讲述 AppEngine 的基本原理。

AppEngine 的数据存储系统构建在 BigTable 数据库之上。BigTable 数据库是建立在 Google 文件系统和其他应用程序之上的，一种压缩的且高性能的专有数据库系统。它采用了本书第四章所介绍的 key/value 数据模型，提供了 Java 数据对象 (JDO) 和 Java 持久 API (JPA) 两种访问接口的实现。这两种访问接口的功能基本上是一样的。下面，我们结合一个来自 AppEngine 网站上的具体实例 (<http://code.google.com/intl/zh-CN/appengine/docs/java/datastore/overview.html>)，讲解一下如何采用 JDO 来访问 BigTable 数据库。该实例将首先定义一个员工 (Employee) 对象，然后通过 JDO 接口来将该对象持久化到 BigTable 数据库中。该实例的部分代码如下：

代码 6.1

```
// Employee.java
import java.util.Date;
import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.IdentityType;
import javax.jdo.annotations.PersistenceCapable;
import javax.jdo.annotations.Persistent;
import javax.jdo.annotations.PrimaryKey;

@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class Employee {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Long id;
```

```
@Persistent
private String firstName;

@Persistent
private String lastName;

@Persistent
private Date hireDate;

public Person(String firstName, String lastName, Date hireDate) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.hireDate = hireDate;
}

// Accessors for the fields. JDO doesn't use these, but your application does.

public Long getId() {
    return id;
}

public String getFirstName() {
    return firstName;
}

// ... other accessors...
}

// PMF.java
import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManagerFactory;

public final class PMF {
    private static final PersistenceManagerFactory pmfInstance =
        JDOHelper.getPersistenceManagerFactory("transactions-optional");

    private PMF() {}

    public static PersistenceManagerFactory get() {
        return pmfInstance;
    }
}

//使用 PMF 类的接口完成持久化
import java.util.Date;
import javax.jdo.JDOHelper;
```

```

import javax.jdo.PersistenceManager;
import javax.jdo.PersistenceManagerFactory;

import Employee;
import PMF;

// ...
Employee employee = new Employee("Alfred", "Smith", new Date());

PersistenceManager pm = PMF.get().getPersistenceManager();

try {
    pm.makePersistent(employee);
} finally {
    pm.close();
}

```

JDO 数据访问和持久化机制大量使用了 Java 6 所提供的 annotation 机制。例如，在 Employee.java 类的 @PrimaryKey 标注了 id 属性将是 BigTable 数据库中实体 Employee 的主键。@Persistent 则表示后面的属性是需要被持久化到数据库中的。PMF.java 则采用了一个工厂模式，用来获得 PersistenceManager 类的实例。PersistenceManager 类提供了一系列接口来真正对数据库进行操作。例如，通过 makePersistent 接口可以把 Employee 对象持久化到数据库中。

此外，JDO 包括一个称为 JDOQL 的查询接口。它可以用来从数据库中检索相应的实体，如下所示：

代码 6.2

```

import java.util.List;
import Employee;

// ...
String query = "select from " + Employee.class.getName() + " where lastName == 'Smith'";
List<Employee> employees = (List<Employee>) pm.newQuery(query).execute();

```

AppEngine 所基于的数据存储系统支持事务功能。事务是数据库中完成单一逻辑功能的所有操作的集合。这些操作要么全部成功，要么全部失败。如果事务成功执行，那么数据库将会做出相应的变化，但如果事务失败，那么对数据库将不会有任何影响。

代码 6.3

```

import javax.jdo.Transaction;
import ClubMembers;

PersistenceManager pm = ...;

```

```
Transaction tx = pm.currentTransaction();

try {
    tx.begin();

    ClubMembers members = pm.getObjectById(ClubMembers.class, "k12345");
    members.incrementCounterBy(1);
    pm.makePersistent(members);

    tx.commit();
} finally {
    if (tx.isActive()) {
        tx.rollback();
    }
}
```

上面给出了一个使用 JDO 事务 API 递增名为 counter 的字段的示例(本例来自 AppEngine 网站：<http://code.google.com/intl/zh-CN/appengine/docs/java/datastore/transactions.html>)，该字段位于名为 ClubMembers 的对象中。JDO 使用 Transaction 类表示事务，该事务是通过前面所介绍的 PersistenceManager 类创建的。这里需要注意的是，上面所示的例子仅是在一个事务中对一个对象进行修改。如果需要在在一个事务中对多个对象进行修改，就必须首先定义实体组。实体组这一概念被用来声明某一实体与其他实体同属一个组。AppEngine 规定在一个事务中抓取、创建、更新或删除的所有实体都必须位于同一实体组中。

6.4.2 AppEngine 的开发环境和部署过程

Google 提供了 Eclipse 插件来帮助用户在 Eclipse 开发平台上创建、测试和上传 AppEngine 应用程序。该插件同时还提供了一个运行在本地的，模拟 AppEngine 运行环境的测试环境，大大简化了 AppEngine 应用程序的开发和测试过程。本书采用了 Eclipse 3.4 版本和相应的 Google 插件来开发 AppEngine 程序。插件的下载地址为：<http://dl.google.com/eclipse/plugin/3.4>。

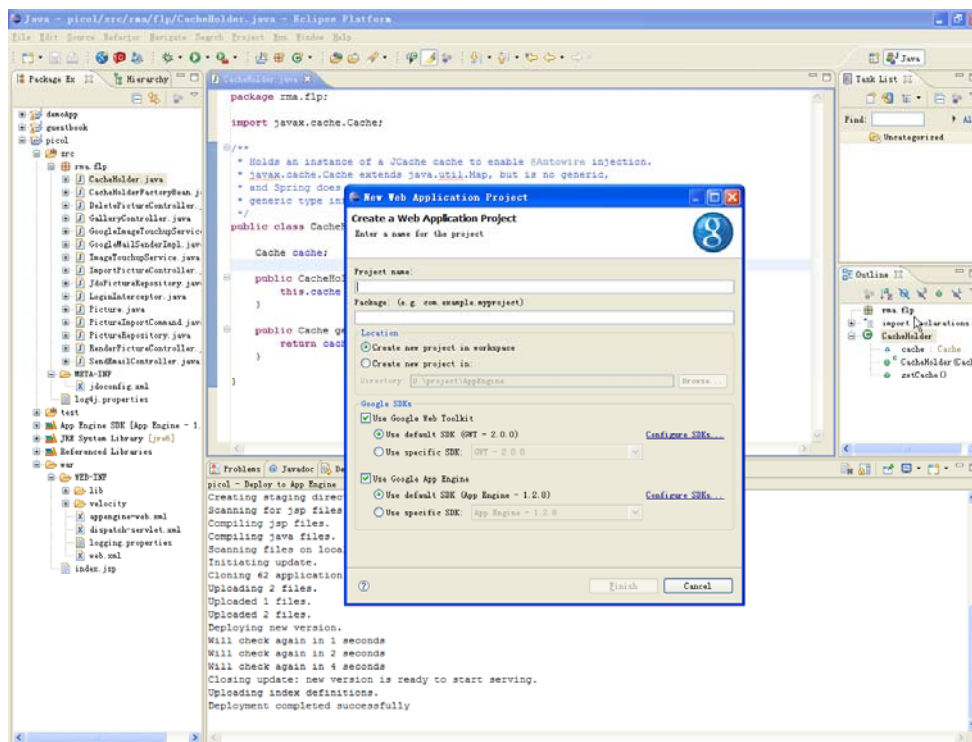


图 6.31 采用 Eclipse 插件的 AppEngine 应用程序开发环境

图 6.31 展示了采用 Eclipse 插件的 AppEngine 应用程序的开发环境。如图所示，当在 Eclipse 环境中安装完 Google 插件后，就会出现一种新的项目类型“Web Application Project”。新建这种类型的项目，就可以开始创建一个全新的 Google AppEngine 应用。

删除的内容: 图

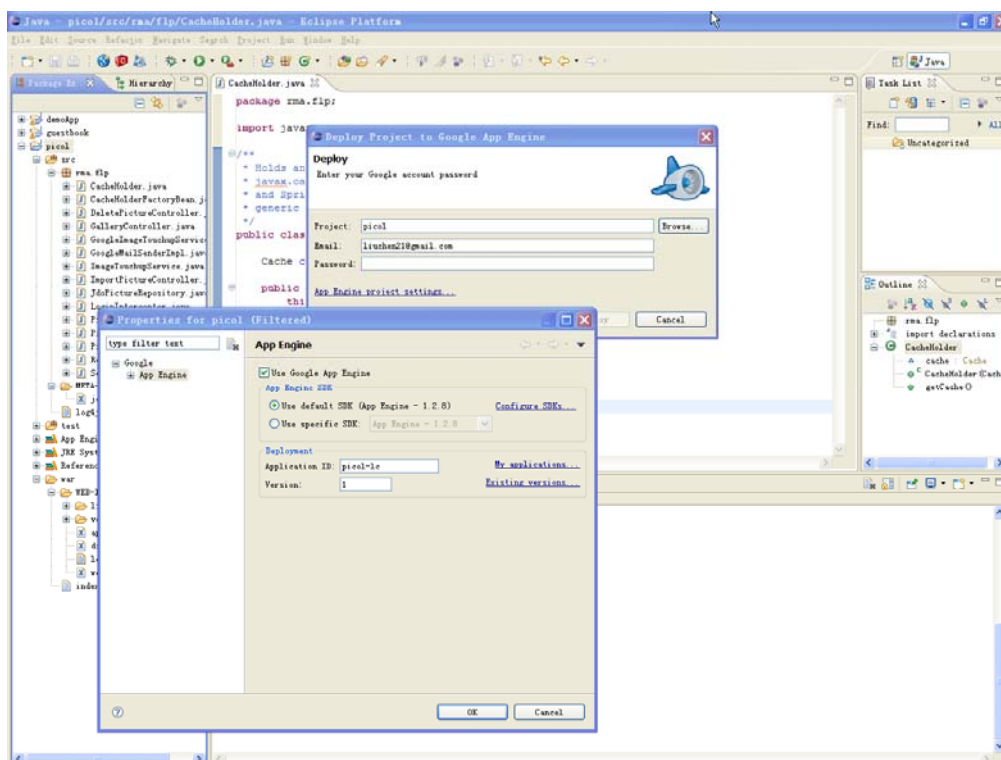


图 6.32 AppEngine 应用的部署

当应用开发完毕后，可以通过 Eclipse 集成开发环境直接将编译通过的应用上传并部署到 Google AppEngine 上加以运行，如图 6.32 所示。需要注意的是，AppEngine 对应用程序定义了版本这一概念。每个上传到 AppEngine 上的应用程序都需要定义它的版本号。版本号是通过阿拉伯数字 1, 2, 3... 等加以区分的。

删除的内容: 图

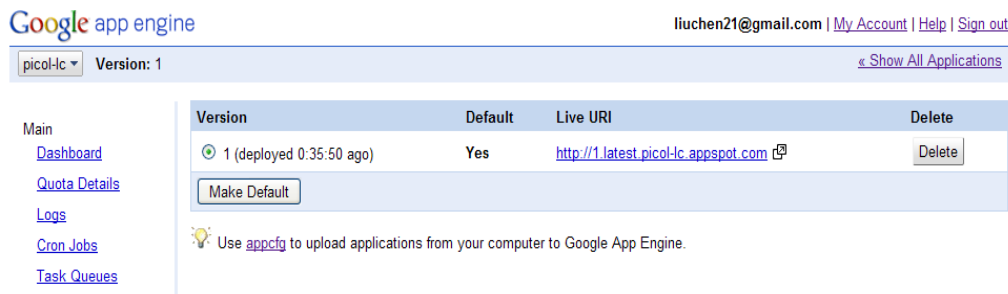


图 6.33 AppEngine 应用的版本管理

图 6.33 展示了 AppEngine 对应用程序的版本管理界面。这个界面以表格的形式列出了一个应用程序的所有版本信息。通过这个界面，用户可以很容易的浏览和管理版本信息。这里需要特别注意的是“Make Default”按钮。这个按钮可以将应用程序的任意一个版本指定为缺省状态。用户通过互联网访问应用程序时，默认访问的就是该应用程序缺省状态的版本。

删除的内容: 图

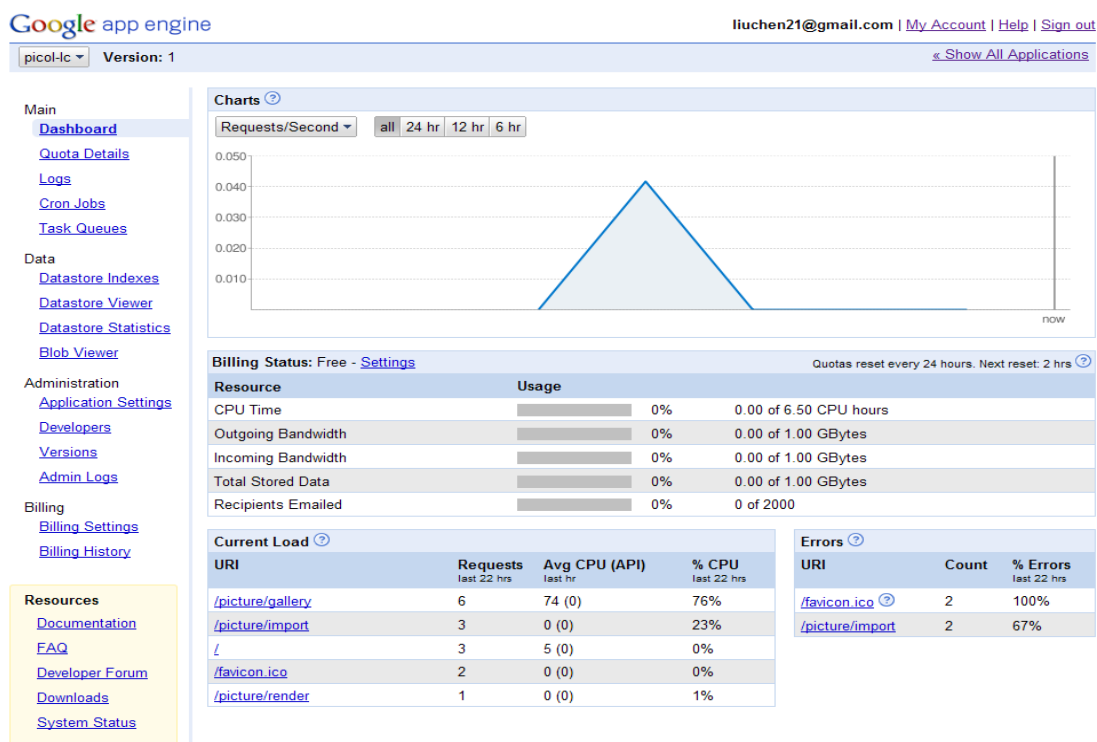


图 6.34 AppEngine 的应用控制台

当应用程序部署完成后，就可以打开 AppEngine 的应用控制台，来管理应用的各种信息。图 6.34 给出了应用控制台的一个简单示例。下面，简单介绍一下应用控制台的一些重要功能。

删除的内容: 图

- (1) Main 菜单: Main 菜单中管理了应用的资源使用情况、日志、计划任务

和任务队列等信息。其中，一个较为重要的功能是 Dashboard（仪表盘）。它记录了应用对各项资源（CPU、带宽和数据存储等）信息的使用情况，并以图表的方式给出了应用在某段时间内的访问情况。

(2) Data 菜单：该菜单项包含了四个子菜单项的功能。通过这些功能，用户可以查看存储在 BigTable 数据库的数据信息，给它们建立索引，同时查看数据存储使用情况的统计信息。图 6.35 中给出了一个使用 Datastore Viewer 菜单项查看应用在数据存储中的数据示例。

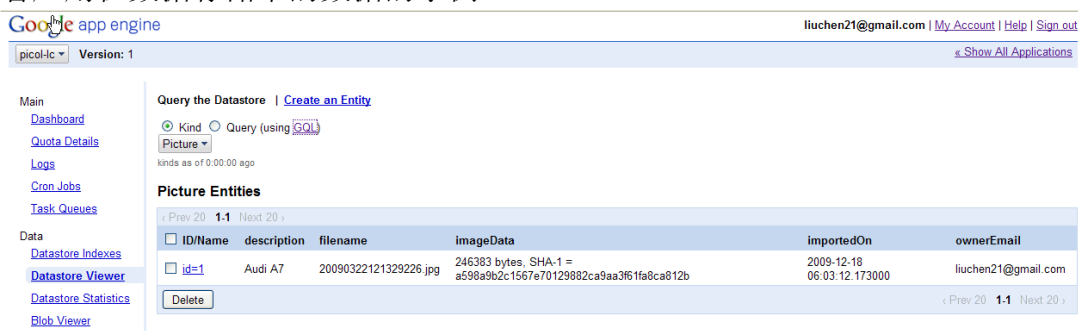


图 6.35 使用 Datastore Viewer 菜单项查看存储的数据

(3) Administration 菜单：该菜单管理了应用的配置信息、开发者信息、版本信息和日志信息等等。其中，版本信息管理是一个比较重要的功能。我们已在图 6.33 展示了这一功能。

(4) Billing 和 Resource 菜单：Billing 菜单为用户提供了进一步订购 AppEngine 资源的功能。对于免费用户，AppEngine 仅提供了有限的资源供他们使用，包括 500MB 的持久存储器和足够的带宽，和每月大约 500 万的页面浏览量。用户可以通过付费，来进一步扩大这些资源的使用量。Resource 菜单则给出了一些对用户开发 AppEngine 应用程序有帮助的文档资料和论坛等，用户可以进一步学习和参考。

6.4.3 AppEngine 的实现

以上介绍了使用 Google AppEngine 进行应用开发和部署的方法，但是，Google AppEngine 的内部架构和实现是什么样的呢？由于 Google AppEngine 既没有公开其内部架构及实现的文档，它本身也非开源产品，本书借助于一个开源的 Google AppEngine 实现——AppScale 来间接地了解 Google AppEngine 的实现原理。

AppScale 可以在云设施之上透明执行（不需要修改的情况下）GAE 的应用。所谓云设施，是指基于虚拟机 Xen 的系统，以及一些 IaaS 系统，例如亚马逊 EC2、S3 和 Eucalyptus 等。AppScale 实现了 GAE 的 Open API，提供了类似于 GAE 提供的工具集。图 6.36 是 AppScale 的体系结构。它包括三类主要的组件：AS、

ALB 和 DBMS。AppController(AC)负责组件之间的通信。

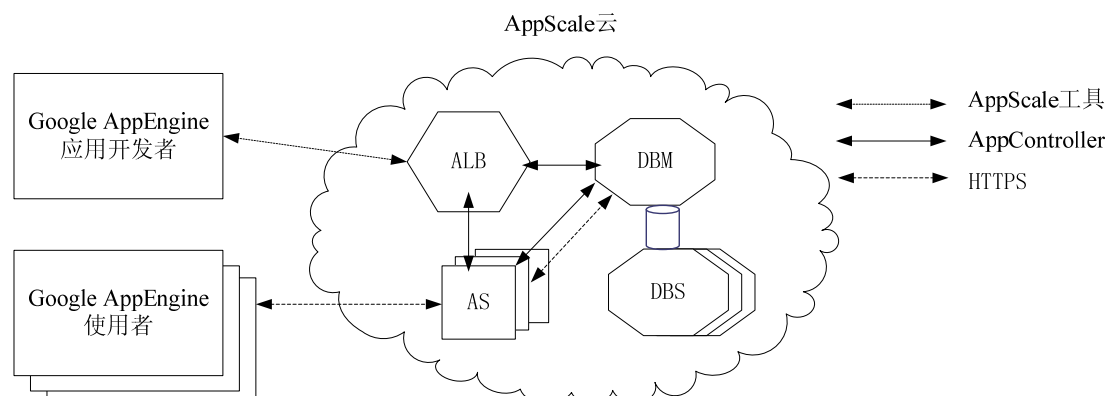


图 6.36 AppScale 的体系结构

一个 AppScale 节点是一个客户虚拟机 (guestVM) 映像的实例，客户虚拟机可以是运行在 Xen 之上的 Linux 系统或者基于 Xen 的云系统(例如亚马逊 EC2 和 Eucalyptus) 等。一个 node 中包括多个组件，组件之间通过 AC 实现交互和通讯。

AppController(AC)负责管理 AppScale 的实例、跨组件的交互以及 GAE 应用的部署。在 AppScale 启动时，由头节点 (head node) 上的 AC 首先启动 ALB，初始化部署和启动其它 guestVM，与这些 guestVM 的节点 AC 通信。头节点的 AC 再启动 DBM 和 DBS 以及 AppServers，并配置每个节点的 IP。

AppLoadBalancer (ALB)在一个部署的头节点上，负责初始化 GAE 应用在 AS 中的连接。用户登陆最初是在 ALB 处理的，ALB 认证后随机选择 AS，然后 ALB 分发用户 GAE 应用的初始请求至选定的 AS。

AppServer(AS)是执行引擎，通过 Https 来与 DBM 互交。AC 在 AS 启动时将数据库位置上报头节点。一个 AS 一次只执行一个 GAE 的应用，一个应用可以使用多个彼此隔离的 GAE。

AppScale 中的数据管理由一个 Database Master (DBM), 多个 Database Slaves (DBSs)构成。DBM 会在 DBS 上存三个副本，支持的操作有：put、get、query 和 delete 等。DBM 节点上的 AC 提供了对于数据存储的接口，ALB 用数据存储来存放开发者上传的 GAE 应用。

AppScale 提供的工具集包括 AppScale 实例的部署和卸载工具、将 GAE 应用上传到一个 AppScale 运行实例上的工具以及通过 AC 查询 AC 和 AS 上的 CPU 和内存占用率的资源使用情况监控工具等。AppScale 提供了较高的容错功能，但与 Google AppEngine 相比，其可伸缩性还略差。

6.5 本章小结

近几年来，随着云计算等互联网计算模式的兴起，XaaS 模式的第三方运营正在得到越来越广泛的关注和认同，XaaS 模式第三方运营和优化已经成为互联网计算的一个重要组成部分。本章首先对 XaaS 模式第三方运营和优化的定义，然后根据运营平台是否对第三方开放，以及根据对互联网应用开发工作控制强度的不同对互联网应用的 XaaS 模式进行了分类。接下来，与 SOA 的“三角架构”概念模型相类比，本章总结了一种 XaaS 模式的概念模型，并分析了 XaaS 模式下应用软件的体系结构。本章还总结归纳了 XaaS 模式几个基本特征：多租户、多租户共享和优化、可伸缩性、可用性和可靠性以及可配置的特性。最后，本章以 Google AppEngine 为例，介绍了一种 XaaS 模式第三方运营的典型实例。